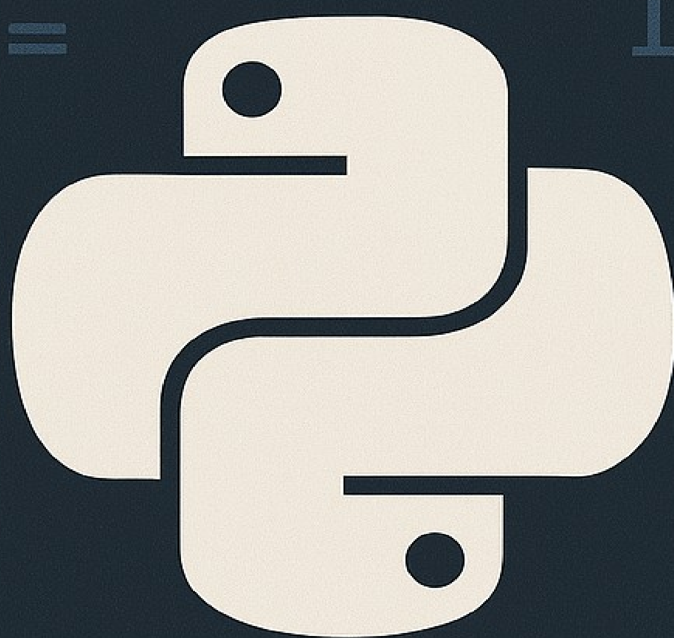
INICIACIÓN

# Python



Luis Minguillón Pascual

# Python

## Iniciación

Luis Mingullón Pascual

# Prefacio

La programación es una de las herramientas más poderosas del mundo moderno. No solo permite crear soluciones a problemas complejos, sino que también abre las puertas a nuevas formas de pensar, analizar y construir. En este contexto, Python se ha convertido en uno de los lenguajes más accesibles y versátiles para quienes desean iniciarse en este apasionante mundo.

Este libro es el primero de una serie de dos volúmenes pensados para acompañarte en tu camino de aprendizaje. **"Iniciación Python"** está diseñado para quienes no tienen conocimientos previos de programación, ofreciendo una introducción clara, práctica y progresiva. El segundo volumen, **"Ejercicios Python"**, complementa y refuerza lo aprendido mediante una amplia colección de ejercicios prácticos que te permitirán afianzar tus habilidades y continuar avanzando con confianza.

Estos libros han sido creados con el objetivo de hacer la programación en Python accesible y amena para todos, independientemente de su punto de partida. Detrás de cada página hay muchas horas de trabajo, reflexión y dedicación, pero sobre todo, hay un gran deseo de compartir conocimiento y facilitar el aprendizaje.

Nada de esto habría sido posible sin el apoyo incondicional de mi familia, que ha sido un pilar fundamental durante todo este proceso. En especial, quiero agradecer profundamente a mi mujer, **Marisa**, por su paciencia, comprensión y aliento constante. Su apoyo ha sido esencial para que este proyecto viera la luz.

Espero que estos libros sean para ti una guía útil y motivadora en tu camino por el mundo de la programación.

¡Bienvenido a la aventura de programar con Python!

Luis Minguillón Pascual

**COPYRICHT**

Indice:

# 1. Variables y Operadores

## Introducción

¡Bienvenido al mundo de la programación con Python! Dos de los conceptos más fundamentales que necesitas dominar desde el principio son las **variables** y los **operadores**.

Piensa en la programación como dar instrucciones a una computadora.

- Las **variables** son como cajas con etiquetas donde guardamos información para usarla más tarde.
- Los **operadores** son las acciones que realizamos con esa información (sumar, comparar, etc.).

¡Vamos a desglosarlos paso a paso!

---

## Capítulo 1: Variables

### 1.1. ¿Qué es una Variable?

Una variable es un espacio en la memoria del ordenador al que le asignamos un nombre (una etiqueta) para almacenar un valor. Este valor puede ser un número, un texto, o tipos de datos más complejos.

La gran ventaja es que, en lugar de recordar un dato específico (como el número 3.14159), podemos simplemente recordar el nombre que le dimos (como pi).

**Analogía:** Imagina una caja de mudanza. Le pones una etiqueta que dice "Libros de Cocina". La caja es la variable, "Libros de Cocina" es el nombre de la variable, y los libros que metes dentro son el valor.

### 1.2. Creando Variables en Python

Crear (o "declarar") una variable en Python es muy sencillo. Solo necesitas un nombre, el signo de igual (=) y el valor que quieres asignarle.

**Sintaxis:**
nombre_de_la_variable = valor

**Ejemplos:**

```
      # Una variable para guardar una edad (número entero)
edad = 25

# Una variable para guardar un precio (número con decimales)
precio_del_cafe = 1.50

# Una variable para guardar un saludo (texto)
mensaje_bienvenida = "Hola, mundo!"
```

```
# Una variable para guardar un estado (verdadero o falso)
esta_lloviendo = False
```

## 1.3. Reglas y Buenas Prácticas para Nombrar Variables

Python tiene algunas reglas para los nombres de las variables:

1. Deben empezar con una letra (a-z, A-Z) o un guion bajo (_).
2. No pueden empezar con un número.
3. Solo pueden contener caracteres alfanuméricos (letras y números) y guiones bajos.
4. Son **sensibles a mayúsculas y minúsculas** (edad es diferente de Edad).

**Buenas Prácticas (Convenciones):**

- **Usa nombres descriptivos:** nombre_cliente es mucho mejor que nc.
- **Usa snake_case:** Es el estilo estándar en Python. Consiste en escribir todo en minúsculas y separar las palabras con guiones bajos.
  - **Bien:** tasa_de_interes_anual
  - **Mal:** TasaDeInteresAnual (esto es CamelCase, más común en otros lenguajes) o tasadeinteresanual.

## 1.4. Tipos de Datos Básicos

Las variables pueden contener diferentes tipos de datos. Los más comunes son:

- **Integer (int):** Números enteros, sin decimales.

  ```
  numero_de_estudiantes = 100
  ```

- **Float (float):** Números de punto flotante, es decir, con decimales.

  ```
  pi = 3.14159
  temperatura = -5.5
  ```

- **String (str):** Cadenas de texto. Siempre van entre comillas simples (') o dobles (").

  ```
  nombre = "Ana"
  frase = 'Python es divertido'
  ```

- **Boolean (bool):** Representan un valor de verdad: True (verdadero) o False (falso). Son fundamentales para la lógica del programa.

  ```
  usuario_registrado = True
  compra_finalizada = False
  ```

## 1.5. Tipado Dinámico

Python es un lenguaje de **tipado dinámico**. Esto significa que no necesitas declarar el tipo de dato de una variable. Python lo infiere automáticamente cuando le asignas un valor. Además, puedes cambiar el tipo de dato de una variable reasignándole un valor diferente.

```
mi_variable = 10         # Python sabe que es un entero (int)
print(type(mi_variable)) # <class 'int'>

mi_variable = "Ahora soy texto" # Ahora la misma variable es un string (str)
print(type(mi_variable)) # <class 'str'>
```

La función type() es muy útil para saber qué tipo de dato contiene una variable.

---

# Capítulo 2: Operadores

Los operadores son símbolos especiales que le dicen a Python que realice una operación matemática, lógica o de comparación.

## 2.1. Operadores Aritméticos

Son los que usas para hacer cálculos matemáticos.

| Operador | Nombre | Ejemplo | Resultado |
|---|---|---|---|
| + | Suma | 10 + 5 | 15 |
| - | Resta | 10 - 5 | 5 |
| * | Multiplicación | 10 * 5 | 50 |
| / | División | 10 / 5 | 2.0 |
| // | División Entera | 10 // 3 | 3 |
| % | Módulo (Resto) | 10 % 3 | 1 |
| ** | Exponenciación | 10 ** 2 | 100 |

**Ejemplo en código:**

```
a = 15
b = 4

print(f"Suma: {a + b}")            # Suma: 19
print(f"Resta: {a - b}")           # Resta: 11
print(f"División: {a / b}")        # División: 3.75 (siempre devuelve un float)
print(f"División Entera: {a // b}") # División Entera: 3 (la parte entera del
resultado)
print(f"Módulo: {a % b}")          # Módulo: 3 (el resto de la división 15/4)
print(f"Potencia: {b ** 2}")       # Potencia: 16 (4 elevado a 2)
```

## 2.2. Operadores de Asignación

Se utilizan para asignar valores a las variables. El más simple es =, pero hay atajos muy útiles.

| Operador | Ejemplo | Equivalente a |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |

**Ejemplo en código:**

```
contador = 0
contador += 1  # Incrementa el contador en 1. Ahora es 1.
print(f"Contador: {contador}")

total = 100
total -= 25    # Decrementa el total en 25. Ahora es 75.
print(f"Total: {total}")
```

## 2.3. Operadores de Comparación

Se usan para comparar dos valores. El resultado de una comparación es siempre un valor booleano (True o False).

| Operador | Nombre | Ejemplo | Resultado |
|---|---|---|---|
| == | Igual que | a == b | False |
| != | No igual que (distinto) | a != b | True |
| > | Mayor que | a > b | True |
| < | Menor que | a < b | False |
| >= | Mayor o igual que | a >= 15 | True |
| <= | Menor o igual que | b <= 5 | True |

**Ejemplo en código:**

```
edad_minima = 18
edad_persona = 22

print(f"¿La persona es mayor de edad? {edad_persona >= edad_minima}") # ¿La
persona es mayor de edad? True
print(f"¿Las edades son iguales? {edad_persona == edad_minima}")     # ¿Las
edades son iguales? False
```

## 2.4. Operadores Lógicos

Estos operadores se usan para combinar expresiones booleanas.

| Operador | Nombre | Descripción |
|---|---|---|
| and | Y lógico | Devuelve True si **ambas** expresiones son True. |
| or | O lógico | Devuelve True si **al menos una** expresión es True. |
| not | Negación | Invierte el valor booleano (True a False y viceversa). |

**Ejemplo en código:**

```
edad = 25
tiene_licencia = True

# ¿Puede conducir un coche alquilado? (Necesita ser mayor de 21 Y tener
licencia)
puede_conducir = (edad > 21) and (tiene_licencia == True)
print(f"¿Puede conducir? {puede_conducir}") # ¿Puede conducir? True

# ¿Tiene acceso a un descuento? (Ser estudiante O ser mayor de 65)
es_estudiante = False
es_mayor_65 = False
tiene_descuento = es_estudiante or es_mayor_65
print(f"¿Tiene descuento? {tiene_descuento}") # ¿Tiene descuento? False

# Negar un booleano
sistema_activo = True
sistema_inactivo = not sistema_activo
print(f"¿El sistema está inactivo? {sistema_inactivo}") # ¿El sistema está
inactivo? False
```

# Poniéndolo Todo Junto: Un Ejemplo Práctico

Imagina que queremos calcular el coste total de una compra en una tienda online.

```
# --- VARIABLES ---
# Definimos los datos de nuestro producto
nombre_producto = "Auriculares Inalámbricos"
precio_unitario = 49.95
cantidad = 2
coste_envio = 7.50

# --- OPERACIONES ---
# Calculamos el subtotal usando un operador aritmético (*)
subtotal = precio_unitario * cantidad

# Calculamos el coste total usando un operador aritmético (+) y de asignación
(+=)
total = subtotal
total += coste_envio # Es lo mismo que total = subtotal + coste_envio

# --- COMPARACIONES Y LÓGICA ---
# Comprobamos si el envío es gratis (si el subtotal supera los 100)
envio_gratis = subtotal >= 100

# Si el envío es gratis, ajustamos el total.
# (¡Esto es un adelanto de las condicionales, que son el siguiente paso lógico!)
if envio_gratis == True:
    total = subtotal # Si el envío es gratis, el total es solo el subtotal

# --- MOSTRAR RESULTADOS ---
print(f"Producto: {nombre_producto}")
print(f"Subtotal: {subtotal} €")
print(f"¿Envío gratis?: {envio_gratis}")
print("------------------------")
print(f"TOTAL A PAGAR: {total} €")
```

# Resumen

- **Variables:** Cajas con etiquetas para guardar datos (nombre = "Python"). Son la base para manejar información.
- **Tipos de Datos:** Los datos pueden ser números (int, float), texto (str) o valores de verdad (bool).
- **Operadores Aritméticos:** Para hacer matemáticas (+, -, *, /).
- **Operadores de Comparación:** Para comparar valores (==, !=, >, <). Devuelven True o False.
- **Operadores Lógicos:** Para combinar condiciones (and, or, not).

# 2. Estructuras Condicionales

## Introducción: Tomando Decisiones en el Código

En la vida real, constantemente tomamos decisiones basadas en condiciones: "Si está lloviendo, cogeré el paraguas. Si no, llevaré gafas de sol". La programación no es diferente. Necesitamos que nuestros programas puedan reaccionar a distintas situaciones y ejecutar diferentes acciones según se cumplan o no ciertas condiciones.

Las **estructuras condicionales** son las herramientas que nos permiten hacer exactamente eso. En Python, las palabras clave para controlar este flujo son if, elif y else.

---

## 1. La Estructura if: La Decisión Básica

El if (que significa "si" en inglés) es la estructura condicional más fundamental. Evalúa una condición y, si es verdadera (True), ejecuta un bloque de código.

### Sintaxis

```
if condicion:
    # Este bloque de código se ejecuta SI Y SÓLO SI la condición es True
    # ¡La indentación (el espacio al principio) es OBLIGATORIA!
    accion_a_realizar()
```

**Puntos Clave:**

1. **if**: La palabra clave que inicia la estructura.
2. **condicion**: Una expresión que da como resultado True (verdadero) o False (falso).
3. **:**: Dos puntos al final de la línea del if. Son imprescindibles.
4. **Indentación**: El código que pertenece al if debe estar "indentado" (generalmente con 4 espacios). Así es como Python sabe qué código depende de la condición.

### Ejemplo 1:

```
  edad = 20

if edad >= 18:
    print("Eres mayor de edad.")
    print("Puedes votar y obtener el carnet de conducir.")

print("Este mensaje se muestra siempre, sin importar la edad.")
```

**¿Cómo funciona?**

- El programa comprueba si edad >= 18 (20 >= 18), lo cual es True.
- Como la condición es verdadera, ejecuta todo el bloque de código indentado.
- Finalmente, ejecuta la línea que está fuera del if.

Si cambiamos edad = 15, la condición 15 >= 18 sería False y los dos primeros print no se ejecutarían.

---

## 2. La Estructura if-else: El Plan B

A menudo, no solo queremos hacer algo si una condición es verdadera, sino también hacer otra cosa si es falsa. Para eso usamos else (que significa "si no").

### Sintaxis

```
if condicion:
    # Bloque de código si la condición es True
    accion_A()
else:
    # Bloque de código si la condición es False
    accion_B()
```

### Ejemplo 2:

```
 temperatura = 15

if temperatura > 25:
    print("Hace calor, ¡pon el aire acondicionado!")
else:
    print("La temperatura es agradable, no necesitas el aire.")
```

**¿Cómo funciona?**

- Python evalúa temperatura > 25 (15 > 25), lo cual es False.
- Como es False, ignora el bloque del if y salta directamente al bloque del else.
- Ejecuta el print dentro del else.

El else garantiza que **siempre** se ejecutará uno de los dos bloques, pero nunca ambos.

---

# 3. La Estructura if-elif-else: Múltiples Caminos

¿Y si tenemos más de dos posibilidades? Por ejemplo, un número puede ser positivo, negativo o cero. Para encadenar varias comprobaciones, usamos elif (contracción de "else if").

## Sintaxis

```
if condicion_1:
    # Bloque si la condicion_1 es True
    accion_A()
elif condicion_2:
    # Bloque si la condicion_1 es False, PERO la condicion_2 es True
    accion_B()
elif condicion_3:
    # Y así sucesivamente...
    accion_C()
else:
    # Bloque si NINGUNA de las condiciones anteriores fue True
    accion_final()
```

**Importante:** Python evalúa las condiciones en orden. En cuanto encuentra una que es True, ejecuta su bloque y omite el resto de elif y el else.

## Ejemplo 3:

```
nota = 7.5

if nota >= 9:
    print("Sobresaliente")
elif nota >= 7:
    print("Notable")
elif nota >= 5:
    print("Aprobado")
else:
    print("Suspenso")
```

**¿Cómo funciona?**

1. ¿nota >= 9? (7.5 >= 9) -> False. Pasa a la siguiente.
2. ¿nota >= 7? (7.5 >= 7) -> True. Ejecuta print("Notable") y **se detiene aquí**. No comprobará el resto.

---

# 4. Construyendo Condiciones: Operadores

Para que las condiciones funcionen, necesitas usar operadores que devuelvan True o False.

## Operadores de Comparación

| Operador | Significado | Ejemplo | Resultado |
|---|---|---|---|
| == | Igual a | 5 == 5 | True |
| != | Distinto de | 5 != 6 | True |
| < | Menor que | 5 < 6 | True |
| > | Mayor que | 6 > 5 | True |
| <= | Menor o igual que | 5 <= 5 | True |
| >= | Mayor o igual que | 6 >= 5 | True |

## Operadores Lógicos

Se usan para combinar varias condiciones.

- **and (y):** Devuelve True si **ambas** condiciones son verdaderas.
- **or (o):** Devuelve True si **al menos una** de las condiciones es verdadera.
- **not (no):** Invierte el resultado de una condición (de True a False y viceversa).

## Ejemplo 4: Combinando operadores

```
edad = 25
tiene_carnet = True

# Usando 'and'
if edad >= 18 and tiene_carnet == True:
    print("Puedes alquilar este coche.")

dia = "Sábado"

# Usando 'or'
```

```
if dia == "Sábado" or dia == "Domingo":
    print("¡Es fin de semana!")

# Usando 'not'
esta_lloviendo = False

if not esta_lloviendo:
    print("No está lloviendo, ¡puedes salir a pasear!")
```

---

# 5. Conceptos Avanzados y Buenas Prácticas

## Condicionales anidados

Puedes poner un if dentro de otro. Esto es útil para comprobaciones más detalladas, pero ten cuidado de no complicar demasiado el código.

```
edad = 22
pais = "España"

if edad >= 18:
    print("Eres mayor de edad.")
    if pais == "España":
        print("Y eres residente en España.")
    else:
        print("Pero no eres residente en España.")
else:
    print("Eres menor de edad.")
```

## Operador Ternario (Expresión Condicional)

Es una forma compacta de escribir un if-else simple en una sola línea. Se usa principalmente para asignar un valor a una variable.

**Sintaxis:** valor_si_true if condicion else valor_si_false

```
# Forma tradicional
edad = 19
```

```
if edad >= 18:
    mensaje = "Mayor de edad"
else:
    mensaje = "Menor de edad"
print(mensaje) # Salida: Mayor de edad

# Con operador ternario
mensaje_ternario = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(mensaje_ternario) # Salida: Mayor de edad
```

IGNORE_WHEN_COPYING_START
content_copy download

## Valores "Truthy" y "Falsy"

En Python, no solo True y False se evalúan en un if. Otros valores también tienen una equivalencia booleana:

- **Valores "Falsy" (se comportan como False):**
  - El número 0 (y 0.0)
  - Una cadena de texto vacía ""
  - Una lista vacía []
  - Una tupla vacía ()
  - Un diccionario vacío {}
  - El valor None
- **Valores "Truthy" (se comportan como True):**
  - Cualquier número distinto de cero (positivo o negativo).
  - Cualquier cadena de texto no vacía.
  - Cualquier lista, tupla o diccionario con al menos un elemento.

Esto permite escribir código más conciso:

```
mi_lista = []

# Forma larga
if len(mi_lista) > 0:
    print("La lista tiene elementos.")
else:
```

```
    print("La lista está vacía.")


# Forma "Pythónica" (usando Truthiness)
if mi_lista:
    print("La lista tiene elementos.")
else:
    print("La lista está vacía.") # Esta línea se ejecutará
```

---

# Resumen Final

- Usa **if** para ejecutar código si una condición es verdadera.
- Usa **if-else** para elegir entre dos bloques de código.
- Usa **if-elif-else** para elegir entre múltiples bloques de código.
- La **indentación** es crucial y define qué código pertenece a cada bloque.
- Combina condiciones con **and** y **or** para crear lógica más compleja.
- Aprovecha los valores **"Truthy" y "Falsy"** para escribir código más limpio y eficiente.

# 3. Estructuras Repetitivas (Bucles)

## 1. Introducción: ¿Qué es un Bucle y por qué es Importante?

En programación, un bucle (o estructura repetitiva) es una instrucción que permite ejecutar un bloque de código múltiples veces. Imagina que necesitas imprimir los números del 1 al 100. Sin un bucle, tendrías que escribir 100 líneas de código. Con un bucle, puedes hacerlo en 3 líneas.

Los bucles son un pilar fundamental de la programación porque nos permiten:

- **Automatizar tareas repetitivas.**
- **Recorrer colecciones de datos** (como listas, diccionarios, etc.).
- **Ahorrar tiempo y reducir la cantidad de código**, siguiendo el principio **DRY** (Don't Repeat Yourself - No te repitas).

Python ofrece dos tipos principales de bucles: el bucle for y el bucle while.

---

## 2. El Bucle for

El bucle for se utiliza para **iterar sobre una secuencia** de elementos. Una "secuencia" puede ser una lista, una tupla, un diccionario, un conjunto, una cadena de texto o un objeto range.

La sintaxis es muy legible y directa:

```
for variable in secuencia:
    # Bloque de código que se ejecuta en cada iteración
    # 'variable' tomará el valor de cada elemento de la 'secuencia'
```

### 2.1. Iterando sobre una Lista

Este es el uso más común. El bucle recorrerá cada elemento de la lista.

```
nombres = ["Ana", "Luis", "Marta", "Javier"]
print("Lista de invitados:")
for nombre in nombres:
    print(f"- {nombre}")
# Salida:
# Lista de invitados:
# - Ana
# - Luis
# - Marta
# - Javier
```

## 2.2. Usando range() para Repetir un Número Fijo de Veces

Si necesitas ejecutar un bloque de código un número específico de veces, la función range() es tu mejor aliada.

- range(stop): Genera números desde 0 hasta stop - 1.
- range(start, stop): Genera números desde start hasta stop - 1.
- range(start, stop, step): Genera números desde start hasta stop - 1, saltando de step en step.

**Ejemplo 1: Imprimir números del 0 al 4**

```
      for i in range(5):
    print(i)

# Salida:
# 0
# 1
# 2
# 3
# 4
```

**Ejemplo 2: Tabla de multiplicar del 7**

```
print("Tabla del 7:")
for i in range(1, 11):  # Del 1 al 10
    resultado = 7 * i
    print(f"7 x {i} = {resultado}")

# Salida:
# Tabla del 7:
# 7 x 1 = 7
# 7 x 2 = 14
# ...
# 7 x 10 = 70
```

## 2.3. Iterando sobre una Cadena de Texto (String)

Un string es una secuencia de caracteres, por lo que también se puede recorrer con un bucle for.

```
palabra = "Python"

for letra in palabra:
    print(letra)

# Salida:
# P
# y
# t
# h
# o
# n
```

## 2.4. Usando enumerate() para Obtener Índice y Valor

A veces, necesitas tanto el elemento como su posición (índice) en la secuencia. Para ello, se usa la función enumerate().

```
colores = ["rojo", "verde", "azul"]

for indice, color in enumerate(colores):
    print(f"El color en la posición {indice} es {color}")

# Salida:
# El color en la posición 0 es rojo
# El color en la posición 1 es verde
# El color en la posición 2 es azul
```

3. El Bucle while

El bucle while ejecuta un bloque de código **mientras una condición específica sea verdadera (True)**. A diferencia del for, no siempre se sabe cuántas veces se repetirá.

La sintaxis es:

```
while condicion:
    # Bloque de código que se ejecuta mientras la condición sea True
    # ¡Importante! Algo dentro del bucle debe poder cambiar la condición a False
```

**¡Advertencia!** Si la condición nunca se vuelve False, crearás un **bucle infinito**, lo que bloqueará tu programa. Siempre asegúrate de que haya una forma de salir del bucle.

## 3.1. Ejemplo Básico de Contador

Este ejemplo simula un for i in range(5): usando un while.

```
      contador = 0
while contador < 5:
    print(f"El contador vale: {contador}")
    contador = contador + 1  # o contador += 1
print("Fin del bucle.")
# Salida:
# El contador vale: 0
# El contador vale: 1
# El contador vale: 2
# El contador vale: 3
# El contador vale: 4
# Fin del bucle.
```

Aquí, la línea contador += 1 es crucial para evitar un bucle infinito.

## 3.2. Ejemplo de Menú de Usuario

Un uso muy común del while es para mantener un programa en ejecución hasta que el usuario decida salir.

```
opcion = ""
while opcion.lower() != "salir":
    opcion = input("Escribe un comando ('ayuda', 'info', 'salir'): ")
    if opcion.lower() == "ayuda":
        print("Esta es la sección de ayuda.")
    elif opcion.lower() == "info":
        print("Información del programa v1.0.")
    elif opcion.lower() == "salir":
        print("¡Hasta pronto!")
    else:
        print("Comando no reconocido.")
```

# 4. Controlando el Flujo de un Bucle

Python ofrece sentencias para controlar el comportamiento de los bucles de forma más precisa.

## 4.1. break: Romper el Bucle

La sentencia break **termina el bucle inmediatamente**, sin importar si la condición del while sigue siendo cierta o si quedan elementos en la secuencia del for.

**Ejemplo: Buscar un número en una lista**

```
numeros = [2, 5, 8, 12, 15, 20]


for numero in numeros:
    print(f"Comprobando {numero}...")
    if numero == 12:
        print("¡Lo encontré!")
        break  # Salimos del bucle porque ya no necesitamos seguir buscando
# Salida:
# Comprobando 2...
# Comprobando 5...
# Comprobando 8...
# Comprobando 12...
# ¡Lo encontré!
```

## 4.2. continue: Saltar a la Siguiente Iteración

La sentencia continue **omite el resto del código de la iteración actual** y salta directamente al inicio de la siguiente.

**Ejemplo: Imprimir solo los números pares**

```
for i in range(1, 10):
    if i % 2 != 0:  # Si el número es impar...
        continue     # ...saltamos esta iteración y no ejecutamos el print.
    print(f"Número par encontrado: {i}")


# Salida:
# Número par encontrado: 2
# Número par encontrado: 4
# Número par encontrado: 6
# Número par encontrado: 8
```

## 4.3. La Cláusula else en Bucles

Esta es una característica única de Python. Un bloque else después de un bucle for o while se ejecuta **solo si el bucle termina de forma natural**, es decir, sin haber sido interrumpido por un break.

**Ejemplo: Buscando un elemento (versión con else)**

```
# Caso 1: Se encuentra el elemento y se activa el break
frutas = ["manzana", "pera", "plátano", "uva"]
for fruta in frutas:
    if fruta == "plátano":
        print("Encontré el plátano.")
        break
else:
    # Este bloque NO se ejecuta porque el bucle fue interrumpido por 'break'
    print("No encontré el plátano.")

# Salida del Caso 1:
# Encontré el plátano.

print("-" * 20)

# Caso 2: No se encuentra el elemento y el bucle termina normalmente
for fruta in frutas:
    if fruta == "cereza":
        print("Encontré la cereza.")
        break
else:
    # Este bloque SÍ se ejecuta porque el bucle terminó sin 'break'
    print("No encontré la cereza.")

# Salida del Caso 2:
# --------------------
# No encontré la cereza.
```

Es muy útil para saber si una búsqueda fue exitosa o no.

---

# 5. Bucles Anidados

Puedes poner un bucle dentro de otro. Esto se conoce como bucles anidados. Son útiles para trabajar con estructuras de datos bidimensionales, como matrices o tablas.

**Ejemplo: Crear una tabla de multiplicar**

```
      # El bucle externo maneja las filas (el número base de la tabla)
for i in range(1, 4):
    print(f"\n--- Tabla del {i} ---")
    # El bucle interno maneja las columnas (el multiplicador)
    for j in range(1, 11):
        print(f"{i} x {j} = {i*j}")


# Salida:
# --- Tabla del 1 ---
# 1 x 1 = 1
# 1 x 2 = 2
# ...
# --- Tabla del 2 ---
# 2 x 1 = 2
# 2 x 2 = 4
# ...
# --- Tabla del 3 ---
# ...
```

# 6. Resumen y Buenas Prácticas

| Bucle | Cuándo usarlo | Ejemplo |
| --- | --- | --- |
| **for** | Cuando sabes cuántas veces quieres iterar o quieres recorrer los elementos de una secuencia (lista, string, etc.). | for fruta in lista_frutas: |
| **while** | Cuando quieres repetir un bloque de código mientras una condición sea cierta, sin saber de antemano el número de iteraciones. | while not usuario_ha_salido: |

**Buenas Prácticas:**

1. **Prefiere for sobre while** cuando iteras sobre secuencias. Es más "Pythónico", más claro y menos propenso a errores (como bucles infinitos).
2. **Evita bucles infinitos** con while. Asegúrate siempre de que la condición pueda cambiar a False.
3. **Mantén el cuerpo del bucle simple.** Si el código dentro del bucle es muy complejo, considera moverlo a una función para mejorar la legibilidad.
4. **Usa break y continue con moderación.** Pueden hacer que el código sea más difícil de seguir si se abusa de ellos.
5. **Aprovecha else** para verificar si un bucle se completó sin interrupciones. Es una herramienta poderosa y expresiva.

# 4. Estructuras de Datos - Las Listas

## Introducción: ¿Qué es una Lista?

En programación, a menudo necesitamos agrupar varios elementos en una sola variable. Imagina una lista de la compra, una lista de invitados a una fiesta o los vagones de un tren. En Python, la estructura de datos que nos permite hacer esto se llama **lista** (list).

Una lista en Python tiene tres características clave:

1. **Es ordenada**: Los elementos mantienen el orden en el que los agregaste. El primer elemento siempre será el primero, y el último siempre será el último.
2. **Es mutable**: Puedes cambiar su contenido después de haberla creado. Puedes agregar, eliminar o modificar elementos.
3. **Permite duplicados**: Una lista puede contener el mismo valor más de una vez.

---

# Capítulo 1: Creación de Listas

Hay varias formas de crear una lista en Python.

### 1.1. Crear una lista vacía

Útil cuando quieres empezar una colección y añadirle elementos más tarde.

```
      # Método 1: Usando corchetes
lista_vacia_1 = []

# Método 2: Usando el constructor list()
lista_vacia_2 = list()

print(lista_vacia_1)
print(lista_vacia_2)
```

**Salida:**

```
[]
[]
```

### 1.2. Crear una lista con elementos iniciales

Puedes crear una lista con valores desde el principio, separándolos por comas dentro de los corchetes.

```
# Lista de números enteros
```

```
numeros = [1, 2, 3, 4, 5]

# Lista de cadenas de texto (strings)
frutas = ["manzana", "banana", "cereza"]

# Lista con tipos de datos mezclados (¡es posible!)
mixta = [1, "hola", 3.14, True]

print(numeros)
print(frutas)
print(mixta)
```

**Salida:**

```
[1, 2, 3, 4, 5]
['manzana', 'banana', 'cereza']
[1, 'hola', 3.14, True]
```

---

# Capítulo 2: Acceso a los Elementos

Una vez que tienes una lista, querrás acceder a sus elementos individuales.

## 2.1. Acceso por Índice

Cada elemento en una lista tiene una posición, llamada **índice**. ¡Importante! **La indexación en Python empieza en 0**.

frutas = ["manzana", "banana", "cereza", "naranja"]

| Elemento | "manzana" | "banana" | "cereza" | "naranja" |
|---|---|---|---|---|
| **Índice Positivo** | 0 | 1 | 2 | 3 |
| **Índice Negativo** | -4 | -3 | -2 | -1 |

```
frutas = ["manzana", "banana", "cereza", "naranja"]
# Acceder al primer elemento (índice 0)
print(frutas[0])  # Salida: manzana

# Acceder al tercer elemento (índice 2)
print(frutas[2])  # Salida: cereza

# Acceder al último elemento con índice negativo
print(frutas[-1]) # Salida: naranja

# Acceder al penúltimo elemento
print(frutas[-2]) # Salida: cereza
```

## 2.2. Rebanado (Slicing)

El rebanado te permite obtener un subconjunto de la lista. La sintaxis es lista[inicio:fin:paso].

- inicio: El índice donde empieza el rebanado (incluido). Si se omite, es 0.
- fin: El índice donde termina el rebanado (**no incluido**). Si se omite, es hasta el final.
- paso: El intervalo entre elementos. Si se omite, es 1.

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Obtener los elementos desde el índice 2 hasta el 4 (el 5 no se incluye)
print(numeros[2:5])   # Salida: [2, 3, 4]
```

```
# Desde el inicio hasta el índice 3 (sin incluirlo)
print(numeros[:3])    # Salida: [0, 1, 2]

# Desde el índice 6 hasta el final
print(numeros[6:])    # Salida: [6, 7, 8, 9]

# Obtener toda la lista (una copia)
print(numeros[:])     # Salida: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Obtener elementos con un paso de 2 (de dos en dos)
print(numeros[0:10:2]) # Salida: [0, 2, 4, 6, 8]

# Un truco genial para invertir una lista
print(numeros[::-1])  # Salida: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

---

# Capítulo 3: Modificación de Listas

Como las listas son mutables, podemos cambiar su contenido.

## 3.1. Cambiar un elemento

Puedes cambiar un elemento asignándole un nuevo valor a través de su índice.

```
frutas = ["manzana", "banana", "cereza"]
print("Antes:", frutas)

# Cambiamos "banana" por "kiwi"
frutas[1] = "kiwi"
print("Después:", frutas)
```

**Salida:**

```
Antes: ['manzana', 'banana', 'cereza']
Después: ['manzana', 'kiwi', 'cereza']
```

### 3.2. Cambiar un rango de elementos

También puedes usar el rebanado para modificar varias partes de una lista.

```
numeros = [0, 1, 2, 3, 4, 5]
print("Antes:", numeros)

# Reemplazar los elementos en los índices 1, 2 y 3
numeros[1:4] = [10, 20, 30]
print("Después:", numeros)
```

**Salida:**

```
Antes: [0, 1, 2, 3, 4, 5]
Después: [0, 10, 20, 30, 4, 5]
```

## Capítulo 4: Métodos Comunes de las Listas

Python proporciona funciones integradas (llamadas métodos) para trabajar con listas.

### 4.1. Añadir elementos

- append(elemento): Añade un elemento al **final** de la lista.
- insert(indice, elemento): Inserta un elemento en una posición específica.
- extend(iterable): Añade todos los elementos de otra lista (u otro iterable) al final.

```
letras = ['a', 'b', 'c']

# Añadir 'd' al final
letras.append('d')
print(letras)  # Salida: ['a', 'b', 'c', 'd']

# Insertar 'x' en la posición 1
letras.insert(1, 'x')
print(letras)  # Salida: ['a', 'x', 'b', 'c', 'd']

# Añadir otra lista al final
mas_letras = ['y', 'z']
letras.extend(mas_letras)
print(letras)  # Salida: ['a', 'x', 'b', 'c', 'd', 'y', 'z']
```

## 4.2. Eliminar elementos

- remove(elemento): Elimina la **primera aparición** del elemento especificado.
- pop(indice): Elimina el elemento en el índice dado y lo **devuelve**. Si no se especifica un índice, elimina y devuelve el último elemento.
- clear(): Elimina todos los elementos de la lista.
- del lista[indice]: Una palabra clave de Python para eliminar un elemento (o un rebanado).

```
numeros = [10, 20, 30, 40, 30, 50]

# Eliminar el primer '30' que encuentre
numeros.remove(30)
print(numeros)  # Salida: [10, 20, 40, 30, 50]

# Eliminar y obtener el elemento en el índice 2 ('40')
elemento_eliminado = numeros.pop(2)
print("Elemento eliminado:", elemento_eliminado) # Salida: 40
print("Lista ahora:", numeros) # Salida: [10, 20, 30, 50]
# Eliminar el último elemento
numeros.pop()
print(numeros)  # Salida: [10, 20, 30]

# Usar 'del'
del numeros[0]
print(numeros)  # Salida: [20, 30]

# Vaciar la lista
numeros.clear()
print(numeros)  # Salida: []
```

## 4.3. Métodos de búsqueda y ordenación

- index(elemento): Devuelve el índice de la primera aparición del elemento.
- count(elemento): Devuelve cuántas veces aparece un elemento en la lista.
- sort(): Ordena la lista **in-situ** (modifica la lista original).
- reverse(): Invierte el orden de los elementos **in-situ**.

```
    mixta = ['a', 'b', 'c', 'a', 'd', 'a']
numeros = [5, 1, 4, 2, 3]

# Encontrar el índice de la primera 'c'
print(mixta.index('c'))  # Salida: 2

# Contar cuántas 'a' hay
print(mixta.count('a'))  # Salida: 3

# Ordenar la lista de números (modifica la original)
numeros.sort()
print(numeros)  # Salida: [1, 2, 3, 4, 5]

# Ordenar en orden descendente
numeros.sort(reverse=True)
print(numeros)  # Salida: [5, 4, 3, 2, 1]

# Invertir el orden actual
numeros.reverse()
print(numeros)  # Salida: [1, 2, 3, 4, 5]
```

**Nota:** sort() solo funciona si todos los elementos son comparables entre sí (ej. todos números o todos strings).

---

# Capítulo 5: Operaciones y Funciones Útiles

Además de los métodos, hay operadores y funciones que son muy útiles con las listas.

- **Concatenación (+)**: Une dos listas para crear una nueva.
- **Repetición (*)**: Repite los elementos de una lista un número de veces.
- **Pertenencia (in)**: Comprueba si un elemento está en la lista (devuelve True o False).
- **len()**: Devuelve el número de elementos en la lista.
- **sorted()**: Devuelve una **nueva** lista ordenada, sin modificar la original.

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]

# Concatenación
lista_unida = lista1 + lista2
print(lista_unida)  # Salida: [1, 2, 3, 4, 5, 6]

# Repetición
lista_repetida = [0] * 5
print(lista_repetida) # Salida: [0, 0, 0, 0, 0]

# Pertenencia
frutas = ["manzana", "banana", "cereza"]
print("banana" in frutas) # Salida: True
print("uva" in frutas)    # Salida: False

# Longitud
print(len(frutas)) # Salida: 3

# Función sorted()
numeros = [5, 1, 4, 2, 3]
numeros_ordenados = sorted(numeros)
print("Original:", numeros)           # Salida: [5, 1, 4, 2, 3]
print("Ordenada:", numeros_ordenados) # Salida: [1, 2, 3, 4, 5]
```

# Capítulo 6: Comprensión de Listas (List Comprehensions)

Esta es una forma avanzada pero muy poderosa y "Pythónica" de crear listas. Te permite generar una nueva lista aplicando una expresión a cada elemento de un iterable, opcionalmente con una condición.

**Sintaxis básica:** [expresion for elemento in iterable if condicion]

## Ejemplo 1: Crear una lista de cuadrados

**Forma tradicional:**

```
cuadrados = []
for i in range(10):
    cuadrados.append(i * i)
print(cuadrados)
```

**Con comprensión de listas:**

```
cuadrados_comp = [i * i for i in range(10)]
print(cuadrados_comp)
```

Ambos producen: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

## Ejemplo 2: Filtrar números pares

**Forma tradicional:**

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = []
for num in numeros:
    if num % 2 == 0:
        pares.append(num)
print(pares)
```

**Con comprensión de listas:**

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares_comp = [num for num in numeros if num % 2 == 0]
print(pares_comp)
```

Ambos producen: [2, 4, 6, 8, 10]

La comprensión de listas es más corta, más legible y a menudo más rápida.

---

# Capítulo 7: Buenas Prácticas y Errores Comunes

## 7.1. Copiar listas: El problema de la asignación

Cuando haces lista_b = lista_a, no estás creando una nueva lista. Ambas variables apuntan al **mismo objeto** en memoria.

```
lista_a = [1, 2, 3]
lista_b = lista_a  # ¡CUIDADO! No es una copia.

lista_b.append(4)

print("Lista A:", lista_a) # Salida: [1, 2, 3, 4]
print("Lista B:", lista_b) # Salida: [1, 2, 3, 4]
```

Para crear una copia independiente, usa el método copy() o el rebanado [:].

```
lista_a = [1, 2, 3]

# Opción 1: método copy()
lista_c = lista_a.copy()

# Opción 2: rebanado
lista_d = lista_a[:]

lista_c.append(4)
lista_d.append(5)

print("Lista A original:", lista_a) # Salida: [1, 2, 3]
print("Lista C (copia):", lista_c) # Salida: [1, 2, 3, 4]
print("Lista D (copia):", lista_d) # Salida: [1, 2, 3, 5]
```

## 7.2. No modifiques una lista mientras la recorres

Esto puede causar un comportamiento inesperado.

```
numeros = [1, 2, 3, 4]
for num in numeros:
    if num == 2:
        numeros.remove(num) # ¡Mala práctica!
# El bucle puede saltarse elementos.
```

La forma segura es iterar sobre una copia:

```
numeros = [1, 2, 3, 4]
for num in numeros.copy(): # Iteramos sobre una copia
    if num % 2 == 0:
        numeros.remove(num) # Modificamos la original
print(numeros) # Salida: [1, 3]
```

# 5. Estructuras de Datos - Conjuntos (Sets)

## 1. ¿Qué es un Conjunto (Set)?

Un set en Python es una colección de elementos con tres características fundamentales:

1. **Es desordenada:** Los elementos no se guardan en un orden particular. No puedes acceder a ellos mediante un índice (como en las listas mi_lista[0]).
2. **Es mutable:** Puedes añadir o eliminar elementos después de su creación.
3. **No permite duplicados:** Cada elemento dentro de un conjunto debe ser único. Si intentas añadir un elemento que ya existe, simplemente no se añadirá.

Piensa en un conjunto como una bolsa donde metes objetos únicos: no importa en qué orden los metiste y no puedes tener dos objetos idénticos.

**¿Cuándo usar un conjunto?**

- Cuando necesitas eliminar duplicados de una lista de forma rápida.
- Cuando quieres realizar operaciones matemáticas de conjuntos (unión, intersección, diferencia, etc.).
- Cuando necesitas comprobar la pertenencia de un elemento a una colección de forma muy eficiente (es mucho más rápido que en una lista).

## 2. Creación de Conjuntos

Hay dos formas principales de crear un conjunto:

**a) Usando llaves {}**

Puedes crear un conjunto directamente con llaves, separando los elementos por comas.

```
# Creando un conjunto de números
numeros = {1, 2, 3, 4, 5}
print(numeros)  # Salida: {1, 2, 3, 4, 5} (el orden puede variar)

# Creando un conjunto de strings
frutas = {"manzana", "banana", "cereza"}
print(frutas)   # Salida: {'cereza', 'manzana', 'banana'} (el orden puede variar)

# Python elimina automáticamente los duplicados
conjunto_con_duplicados = {1, 2, 2, 3, 3, 3, 4}
print(conjunto_con_duplicados)  # Salida: {1, 2, 3, 4}
```

**b) Usando la función set()**

Puedes usar el constructor set() para crear un conjunto a partir de cualquier objeto iterable (como una lista, una tupla o un string).

```
# A partir de una lista
lista_numeros = [1, 2, 2, 3, 4]
conjunto_desde_lista = set(lista_numeros)
print(conjunto_desde_lista) # Salida: {1, 2, 3, 4}

# A partir de una tupla
tupla_frutas = ("manzana", "banana", "manzana")
conjunto_desde_tupla = set(tupla_frutas)
print(conjunto_desde_tupla) # Salida: {'banana', 'manzana'}

# A partir de un string (cada carácter se convierte en un elemento)
conjunto_desde_string = set("hola mundo")
print(conjunto_desde_string) # Salida: {'m', 'o', 'l', 'd', 'u', 'h', ' ', 'a', 'n'}
```

**¡Atención! Creando un conjunto vacío**

Para crear un conjunto vacío, **debes** usar set(). Si usas {}, Python creará un diccionario vacío.

```
# Correcto
conjunto_vacio = set()
print(type(conjunto_vacio)) # Salida: <class 'set'>

# Incorrecto (esto crea un diccionario)
diccionario_vacio = {}
print(type(diccionario_vacio)) # Salida: <class 'dict'>
```

# 3. Operaciones Básicas con Conjuntos

**a) Añadir elementos**

- **.add(elemento)**: Añade un único elemento al conjunto.
- **.update(iterable)**: Añade múltiples elementos de un iterable (lista, tupla, otro conjunto).

```
colores = {"rojo", "verde"}

# Añadir un solo elemento
colores.add("azul")
print(colores)  # Salida: {'rojo', 'verde', 'azul'}

# Intentar añadir un elemento existente (no hace nada)
colores.add("rojo")
print(colores)  # Salida: {'rojo', 'verde', 'azul'}

# Añadir múltiples elementos desde una lista
colores.update(["amarillo", "naranja", "verde"])
print(colores)  # Salida: {'naranja', 'verde', 'amarillo', 'rojo', 'azul'}
```

**b) Eliminar elementos**

Hay tres formas principales de eliminar elementos:

- **.remove(elemento)**: Elimina el elemento especificado. Si el elemento no existe, lanza un error (KeyError).
- **.discard(elemento)**: Elimina el elemento especificado. Si el elemento no existe, **no hace nada** (no lanza error). Es más seguro si no estás seguro de que el elemento exista.
- **.pop()**: Elimina y devuelve un elemento **arbitrario** del conjunto. Como los conjuntos son desordenados, no sabes cuál eliminará.
- **.clear()**: Elimina todos los elementos del conjunto, dejándolo vacío.

```
elementos = {10, 20, 30, 40, 50}

# Usando remove
elementos.remove(30)
print(elementos)  # Salida: {10, 20, 40, 50}
# elementos.remove(99) # Esto lanzaría un KeyError

# Usando discard
elementos.discard(40)
print(elementos)  # Salida: {10, 20, 50}
elementos.discard(99) # No pasa nada, no hay error
```

```
print(elementos)  # Salida: {10, 20, 50}

# Usando pop
elemento_eliminado = elementos.pop()
print(f"Elemento eliminado: {elemento_eliminado}")
print(f"Conjunto restante: {elementos}")

# Limpiar el conjunto
elementos.clear()
print(elementos)  # Salida: set()
```

**c) Otras operaciones comunes**

- **len(conjunto)**: Devuelve el número de elementos.
- **elemento in conjunto**: Comprueba si un elemento existe en el conjunto. Devuelve True o False.

```
vocales = {'a', 'e', 'i', 'o', 'u'}
print(f"Número de vocales: {len(vocales)}") # Salida: 5

print('a' in vocales)     # Salida: True
print('b' in vocales)     # Salida: False
print('z' not in vocales) # Salida: True
```

# 4. Operaciones Matemáticas de Conjuntos

Esta es la verdadera potencia de los conjuntos. Permiten realizar operaciones lógicas de forma muy eficiente.

Usemos estos dos conjuntos para los ejemplos:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
```

**a) Unión**

Combina todos los elementos de ambos conjuntos, sin duplicados.

- **Operador:** |
- **Método:** .union()

```
# Usando el operador
union_op = a | b
print(union_op)  # Salida: {1, 2, 3, 4, 5, 6}

# Usando el método
union_metodo = a.union(b)
print(union_metodo) # Salida: {1, 2, 3, 4, 5, 6}
```

**b) Intersección**

Devuelve solo los elementos que están presentes en **ambos** conjuntos.

- **Operador:** &
- **Método:** .intersection()

```
# Usando el operador
inter_op = a & b
print(inter_op) # Salida: {3, 4}

# Usando el método
inter_metodo = a.intersection(b)
print(inter_metodo) # Salida: {3, 4}
```

**c) Diferencia**

Devuelve los elementos que están en el primer conjunto pero **no** en el segundo.

- **Operador:** -
- **Método:** .difference()

```
# Elementos en 'a' pero no en 'b'
diff_a_b = a - b
print(diff_a_b) # Salida: {1, 2}

# Elementos en 'b' pero no en 'a'
```

```
diff_b_a = b.difference(a)
print(diff_b_a) # Salida: {5, 6}
```

**d) Diferencia Simétrica**

Devuelve los elementos que están en uno de los conjuntos, pero **no en ambos**.

- **Operador:** ^
- **Método:** .symmetric_difference()

```
# Usando el operador
sym_diff_op = a ^ b
print(sym_diff_op) # Salida: {1, 2, 5, 6}


# Usando el método
sym_diff_metodo = a.symmetric_difference(b)
print(sym_diff_metodo) # Salida: {1, 2, 5, 6}
```

# 5. Iterar sobre un Conjunto

Puedes recorrer los elementos de un conjunto usando un bucle for, pero recuerda que el orden no está garantizado.

```
mi_conjunto = {"Python", "Java", "C++", "Go"}
for lenguaje in mi_conjunto:
    print(lenguaje)

# Salida (el orden puede ser diferente en cada ejecución):
# Java
# Go
# Python
# C++
```

# 6. Frozenset: El Conjunto Inmutable

Python también ofrece una versión inmutable del conjunto llamada frozenset. Una vez creado, no puedes añadir ni eliminar elementos.

**¿Por qué usar un frozenset?**
Porque al ser inmutable, es "hashable". Esto significa que puedes usar un frozenset como:

- Una clave en un diccionario.
- Un elemento dentro de otro conjunto.

```
# Creando un frozenset
conjunto_congelado = frozenset([1, 2, 3, 2])
print(conjunto_congelado)  # Salida: frozenset({1, 2, 3})


# No se puede modificar
# conjunto_congelado.add(4) # Lanzaría un AttributeError
# Uso como clave de diccionario
diccionario_sets = {frozenset({1, 2}): "valor1", frozenset({3, 4}): "valor2"}
print(diccionario_sets[frozenset({1, 2})]) # Salida: valor1
```

# 7. Tabla Resumen (Cheat Sheet)

| Operación | Método | Operador | Descripción |
|---|---|---|---|
| **Crear vacío** | set() | - | Crea un conjunto vacío. |
| **Añadir uno** | .add(elem) | - | Añade un elemento. |
| **Añadir varios** | .update(iter) | - | Añade múltiples elementos de un iterable. |
| **Eliminar (con error)** | .remove(elem) | - | Elimina un elemento. Lanza KeyError si no existe. |
| **Eliminar (sin error)** | .discard(elem) | - | Elimina un elemento. No hace nada si no existe. |
| **Eliminar aleatorio** | .pop() | - | Elimina y devuelve un elemento arbitrario. |
| **Limpiar** | .clear() | - | Elimina todos los elementos. |
| **Unión** | .union(otro) | \| | Elementos de ambos conjuntos. |
| **Intersección** | .intersection(otro) | & | Elementos comunes en ambos conjuntos. |
| **Diferencia** | .difference(otro) | - | Elementos en el primero pero no en el segundo. |
| **Dif. Simétrica** | .symmetric_difference(otro) | ^ | Elementos que no son comunes. |

---

# 6. Estructuras de Datos - Tuplas

## Índice

---

# 1. ¿Qué son las Tuplas?

### Definición y Características Principales

Una **tupla** es una estructura de datos en Python que permite almacenar una colección ordenada de elementos. Es muy similar a una lista, pero con una diferencia fundamental: **las tuplas son inmutables**.

- **Ordenada**: Los elementos se guardan en el orden en que se definen y este orden no cambia.
- **Permite Duplicados**: Una tupla puede contener elementos idénticos.
- **Heterogénea**: Puede contener elementos de diferentes tipos (números, strings, otras tuplas, listas, etc.).
- **Inmutable**: Una vez creada, no se puede modificar. No puedes agregar, eliminar o cambiar sus elementos.

### Diferencia Clave con las Listas: Inmutabilidad

| Característica | Listas ([]) | Tuplas (()) |
|---|---|---|
| **Sintaxis** | mi_lista = [1, 2, 3] | mi_tupla = (1, 2, 3) |
| **Mutabilidad** | **Mutable** (se puede cambiar) | **Inmutable** (no se puede cambiar) |
| **Uso principal** | Colecciones que necesitan cambiar | Datos que no deben cambiar |

---

# 2. Creación de Tuplas

### Sintaxis Básica

La forma más común de crear una tupla es usando paréntesis () y separando los elementos con comas.

```
# Tupla de números enteros
coordenadas = (10, 20, 30)
print(coordenadas)  # Salida: (10, 20, 30)

# Tupla con diferentes tipos de datos
datos_persona = ("Ana", 25, 1.68, True)
print(datos_persona) # Salida: ('Ana', 25, 1.68, True)

# Tupla vacía
tupla_vacia = ()
print(tupla_vacia) # Salida: ()
```

### El Caso Especial de la Tupla de un Solo Elemento

Para crear una tupla con un único elemento, **debes incluir una coma al final**. De lo contrario, Python lo interpretará como un simple valor entre paréntesis.

```
no_es_una_tupla = (50)
print(type(no_es_una_tupla))  # Salida: <class 'int'>

si_es_una_tupla = (50,)
print(type(si_es_una_tupla))  # Salida: <class 'tuple'>
print(si_es_una_tupla)      # Salida: (50,)
```

### Creación sin Paréntesis (Tuple Packing)

También puedes crear una tupla simplemente asignando una secuencia de valores separados por comas a una variable. Esto se conoce como "empaquetado de tuplas".

```
mis_colores = "rojo", "verde", "azul"
print(mis_colores)       # Salida: ('rojo', 'verde', 'azul')
print(type(mis_colores)) # Salida: <class 'tuple'>
```

### Uso del Constructor tuple()

Puedes crear una tupla a partir de cualquier objeto "iterable" (como una lista, un string o un rango) usando la función tuple().

```
# A partir de una lista
lista = [1, 2, 3]
tupla_desde_lista = tuple(lista)
print(tupla_desde_lista)  # Salida: (1, 2, 3)

# A partir de un string
string = "Hola"
tupla_desde_string = tuple(string)
print(tupla_desde_string) # Salida: ('H', 'o', 'l', 'a')
```

---

## 3. Acceso a los Elementos

El acceso a los elementos de una tupla es idéntico al de las listas, utilizando corchetes [].

**Indexación**

Los índices comienzan en 0 para el primer elemento.

```
dias_semana = ("Lunes", "Martes", "Miércoles", "Jueves", "Viernes")

primer_dia = dias_semana[0]  # "Lunes"
tercer_dia = dias_semana[2]  # "Miércoles"

print(f"El primer día es {primer_dia}")
```

**Indexación Negativa**

Puedes usar índices negativos para acceder a los elementos desde el final. -1 es el último elemento.

```
ultimo_dia = dias_semana[-1]  # "Viernes"
penultimo_dia = dias_semana[-2] # "Jueves"

print(f"El último día es {ultimo_dia}")
```

**Rebanadas (Slicing)**

Permite obtener una sub-tupla especificando un rango [inicio:fin:paso].

```
fin_de_semana = ("Sábado", "Domingo")
dias_laborales = dias_semana[0:5] # No incluye el índice 5
print(dias_laborales) # Salida: ('Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes')
# Desde el inicio hasta el índice 2 (no incluido)
primeros_dos_dias = dias_semana[:2]
print(primeros_dos_dias) # Salida: ('Lunes', 'Martes')

# Desde el índice 3 hasta el final
ultimos_dias = dias_semana[3:]
print(ultimos_dias) # Salida: ('Jueves', 'Viernes')
```

---

## 4. La Inmutabilidad: La Característica Clave

### ¿Qué significa que sea inmutable?

Significa que, una vez creada la tupla, no puedes cambiar su contenido.

```
mi_tupla = (10, 20, 30)

# Esto generará un error
# mi_tupla[0] = 100
# TypeError: 'tuple' object does not support item assignment

# Tampoco puedes añadir o eliminar elementos
# mi_tupla.append(40) # AttributeError: 'tuple' object has no attribute 'append'
# del mi_tupla[0]     # TypeError: 'tuple' object doesn't support item deletion
```

Si necesitas una versión modificada, debes crear una nueva tupla.

```
tupla_original = (1, 2, 3)
nueva_tupla = tupla_original + (4, 5) # La concatenación crea una nueva tupla
print(nueva_tupla) # Salida: (1, 2, 3, 4, 5)
```

### La Excepción: Objetos Mutables Dentro de una Tupla

Si una tupla contiene un objeto mutable (como una lista), **no puedes reemplazar la lista por otro objeto**, pero **sí puedes modificar el contenido de esa lista**.

```
tupla_mixta = (1, "hola", [10, 20])

# Esto NO es posible: no se puede reasignar el elemento en el índice 2
# tupla_mixta[2] = [30, 40] # Genera TypeError

# Pero ESTO SÍ es posible: se puede modificar el contenido de la lista interna
tupla_mixta[2].append(30)
print(tupla_mixta) # Salida: (1, 'hola', [10, 20, 30])
```

---

## 5. Operaciones y Métodos Comunes

### Concatenación (+) y Repetición (*)

```
tupla1 = (1, 2)
tupla2 = (3, 4)
```

```
# Concatenación: crea una nueva tupla uniendo ambas
tupla_suma = tupla1 + tupla2
print(tupla_suma) # Salida: (1, 2, 3, 4)

# Repetición: crea una nueva tupla repitiendo los elementos
tupla_repetida = ("eco",) * 3
print(tupla_repetida) # Salida: ('eco', 'eco', 'eco')
```

**Operadores de Pertenencia (in, not in)**

Verifican si un elemento existe dentro de la tupla.

```
numeros = (10, 20, 30, 40, 50)
print(30 in numeros)      # Salida: True
print(100 not in numeros) # Salida: True
```

**Funciones Integradas (len, max, min, sum)**

```
numeros = (5, 15, 2, 30, 10)
print(len(numeros))   # Salida: 5 (cantidad de elementos)
print(max(numeros))   # Salida: 30 (el valor máximo)
print(min(numeros))   # Salida: 2 (el valor mínimo)
print(sum(numeros))   # Salida: 62 (la suma de los elementos)
```

**Métodos de Tupla: .count() y .index()**

Las tuplas tienen muy pocos métodos propios debido a su inmutabilidad.

- **tupla.count(elemento)**: Devuelve el número de veces que aparece un elemento.
- **tupla.index(elemento)**: Devuelve el índice de la primera aparición del elemento. Si no lo encuentra, genera un error ValueError.

```
mi_tupla = ('a', 'b', 'c', 'a', 'd', 'a')

# Contar cuántas veces aparece 'a'
conteo_a = mi_tupla.count('a')
print(f"La letra 'a' aparece {conteo_a} veces") # Salida: 3
```

```
# Encontrar el índice de 'c'
indice_c = mi_tupla.index('c')
print(f"La letra 'c' está en el índice {indice_c}") # Salida: 2
```

---

## 6. Desempaquetado de Tuplas (Tuple Unpacking)

Esta es una de las características más elegantes y útiles de las tuplas en Python.

### Asignación Múltiple

Puedes asignar los elementos de una tupla a múltiples variables a la vez. El número de variables debe coincidir con el número de elementos.

```
coordenadas = (120, 80)
x, y = coordenadas

print(f"x: {x}, y: {y}") # Salida: x: 120, y: 80

# Esto es muy útil para devolver múltiples valores de una función
def get_user_info():
    return ("Alex", "alex@example.com", 30)

nombre, email, edad = get_user_info()
print(f"Nombre: {nombre}, Email: {email}")
```

### Uso del Asterisco (*) para Desempaquetado Extendido

Si no sabes cuántos elementos habrá o solo te interesan algunos, puedes usar * para agrupar el resto en una lista.

```
numeros = (1, 2, 3, 4, 5, 6)

# Capturar el primero y el resto
primero, *resto = numeros
print(f"Primero: {primero}")  # Salida: 1
print(f"Resto: {resto}")      # Salida: [2, 3, 4, 5, 6] (¡Es una lista!)

# Capturar el primero, el último y los del medio
```

```
primero, *medio, ultimo = numeros
print(f"Primero: {primero}, Último: {ultimo}") # Salida: 1, 6
print(f"Medio: {medio}")                        # Salida: [2, 3, 4, 5]
```

**Aplicaciones Prácticas: Bucles y Funciones**

El desempaquetado es muy común en bucles for, especialmente con la función enumerate(), que devuelve tuplas de (índice, valor).

```
colores = ("rojo", "verde", "azul")

for indice, color in enumerate(colores):
    print(f"Índice {indice}: {color}")
```

---

## 7. ¿Cuándo Usar Tuplas en Lugar de Listas?

La elección entre una tupla y una lista depende de lo que necesites hacer con los datos.

**Tabla Comparativa Rápida**

| Usa una **Tupla** si... | Usa una **Lista** si... |
|---|---|
| Los datos **no deben cambiar** (integridad de datos). | La colección de datos **necesita cambiar** (añadir, quitar, ordenar). |
| Quieres usar la colección como **clave de un diccionario**. | Necesitas métodos específicos de listas como .append(), .sort(), .remove(). |
| Quieres **devolver múltiples valores** de una función. | La colección es homogénea y su tamaño puede variar. |
| Buscas una ligera **mejora de rendimiento** (son más rápidas de iterar y ocupan menos memoria). | La flexibilidad es más importante que la inmutabilidad. |

**Casos de Uso Ideales para Tuplas**

- **Coordenadas geográficas**: (latitud, longitud)
- **Colores RGB**: (255, 0, 128)
- **Registros de una base de datos**: (id_usuario, nombre, fecha_registro)
- **Claves de diccionario compuestas**: puntos = {("Juan", "Nivel1"): 1500, ("Ana", "Nivel2"): 3200}

---

## 8. Resumen

- Las **tuplas** son colecciones **ordenadas e inmutables**.
- Se crean con paréntesis () y los elementos se separan con comas. ¡Recuerda la coma final para tuplas de un solo elemento (elemento,)!
- Su **inmutabilidad** es su rasgo definitorio: garantiza que los datos no se modifiquen accidentalmente.
- Son ideales para datos fijos, como constantes, registros o claves de diccionario.
- El **desempaquetado de tuplas** es una técnica muy poderosa  para escribir código más limpio y eficiente.

# 7. Estructuras de Datos - Diccionarios

## 1. ¿Qué es un Diccionario?

Imagina un diccionario de la vida real. No buscas una palabra por su número de página (como en una lista), sino por la palabra misma (la **clave**) para encontrar su definición (el **valor**).

En Python, un diccionario es exactamente eso: una colección no ordenada de elementos, donde cada elemento es un par **clave-valor**.

**Características principales:**

- **Pares Clave-Valor:** Cada elemento se compone de una clave única y un valor asociado.
- **Rápido Acceso:** Son extremadamente eficientes para buscar, añadir y eliminar elementos a través de su clave.
- **Mutable:** Puedes cambiar su contenido (añadir, modificar o eliminar pares clave-valor) después de su creación.
- **No Ordenado (históricamente):** Antes de Python 3.7, los diccionarios no garantizaban ningún orden. **A partir de Python 3.7, los diccionarios mantienen el orden de inserción**. Esto es una mejora muy importante.
- **Claves Únicas e Inmutables:**
    - Las claves dentro de un diccionario deben ser únicas. Si intentas añadir una clave que ya existe, su valor se sobrescribirá.
    - Las claves deben ser de un tipo de dato inmutable (como strings, números o tuplas). No puedes usar listas o otros diccionarios como claves.

## 2. Creación de un Diccionario

Puedes crear un diccionario de varias formas.

**a) Usando llaves {}:** Es la forma más común.

```
# Un diccionario vacío
diccionario_vacio = {}
print(diccionario_vacio)  # Salida: {}


# Un diccionario con datos iniciales
persona = {
    "nombre": "Ana",
    "edad": 28,
    "ciudad": "Madrid",
    "habilidades": ["Python", "SQL", "Análisis de datos"]
```

```
}
print(persona)
# Salida: {'nombre': 'Ana', 'edad': 28, 'ciudad': 'Madrid', 'habilidades': ['Python', 'SQL', 'Análisis de datos']}
```

**b) Usando la función dict():**

```
# A partir de pares clave-valor
usuario = dict(nombre="Carlos", id=102, activo=True)
print(usuario)
# Salida: {'nombre': 'Carlos', 'id': 102, 'activo': True}

# A partir de una lista de tuplas
configuracion = dict([('tema', 'oscuro'), ('idioma', 'es')])
print(configuracion)
# Salida: {'tema': 'oscuro', 'idioma': 'es'}
```

# 3. Operaciones Básicas (CRUD)

CRUD son las siglas de **C**reate (Crear), **R**ead (Leer), **U**pdate (Actualizar) y **D**elete (Eliminar).

## a) Acceder a los Valores (Leer)

Para acceder al valor asociado a una clave, usas corchetes [].

```
persona = {"nombre": "Ana", "edad": 28, "ciudad": "Madrid"}

# Acceder al valor de la clave "nombre"
print(persona["nombre"])  # Salida: Ana
print(persona["edad"])    # Salida: 28
```

**¡Cuidado!** Si intentas acceder a una clave que no existe, Python lanzará un error KeyError.

```
# print(persona["profesion"]) # Esto daría un KeyError
```

**Forma segura de acceder a los valores: el método .get()**

El método .get() te permite obtener un valor y, si la clave no existe, devuelve None (o un valor por defecto que tú especifiques) en lugar de un error.

```
# La clave "profesion" no existe, devuelve None
profesion = persona.get("profesion")
print(profesion)  # Salida: None

# Especificando un valor por defecto
profesion = persona.get("profesion", "No especificada")
print(profesion)  # Salida: No especificada
```

## b) Añadir o Modificar Elementos (Crear/Actualizar)

La sintaxis es la misma para añadir un nuevo par o para modificar el valor de uno existente.

```
persona = {"nombre": "Ana", "edad": 28}

# Modificar un valor existente
persona["edad"] = 29
print(persona)  # Salida: {'nombre': 'Ana', 'edad': 29}

# Añadir un nuevo par clave-valor
persona["ciudad"] = "Barcelona"
print(persona)  # Salida: {'nombre': 'Ana', 'edad': 29, 'ciudad': 'Barcelona'}
```

## c) Eliminar Elementos (Eliminar)

Hay varias formas de eliminar elementos.

**1. Usando la palabra clave del:** Elimina un par clave-valor específico.

```
coche = {"marca": "Ford", "modelo": "Mustang", "año": 1964}
del coche["año"]
print(coche)  # Salida: {'marca': 'Ford', 'modelo': 'Mustang'}
```

**2. Usando el método .pop():** Elimina un par y **devuelve el valor** eliminado.

```
coche = {"marca": "Ford", "modelo": "Mustang", "año": 1964}
modelo_eliminado = coche.pop("modelo")
```

```
print(f"El modelo eliminado fue: {modelo_eliminado}") # Salida: El modelo eliminado fue: Mustang
print(coche) # Salida: {'marca': 'Ford', 'año': 1964}
```

**3. Usando el método .popitem():** Elimina y devuelve el último par clave-valor insertado (como una tupla).

```
coche = {"marca": "Ford", "modelo": "Mustang", "año": 1964}
ultimo_item = coche.popitem()

print(f"El último par eliminado fue: {ultimo_item}") # Salida: El último par eliminado fue: ('año', 1964)
print(coche) # Salida: {'marca': 'Ford', 'modelo': 'Mustang'}
```

**4. Usando el método .clear():** Elimina todos los elementos del diccionario.

```
coche = {"marca": "Ford", "modelo": "Mustang"}
coche.clear()
print(coche) # Salida: {}
```

# 4. Iterar sobre un Diccionario

Es muy común necesitar recorrer los elementos de un diccionario.

```
persona = {"nombre": "Ana", "edad": 29, "ciudad": "Barcelona"}
```

**a) Iterar sobre las claves (comportamiento por defecto):**

```
for clave in persona:
    print(f"Clave: {clave}")
# Salida:
# Clave: nombre
# Clave: edad
# Clave: ciudad
```

**b) Iterar sobre los valores usando .values():**

```
for valor in persona.values():
    print(f"Valor: {valor}")
# Salida:
# Valor: Ana
# Valor: 29
# Valor: Barcelona
```

**c) Iterar sobre los pares clave-valor usando .items() (la forma más útil):**

```
for clave, valor in persona.items():
    print(f"La clave '{clave}' tiene el valor '{valor}'")
# Salida:
# La clave 'nombre' tiene el valor 'Ana'
# La clave 'edad' tiene el valor '29'
# La clave 'ciudad' tiene el valor 'Barcelona'
```

# 5. Métodos Útiles Adicionales

- **len(diccionario):** Devuelve el número de pares clave-valor.

  ```
  print(len(persona))  # Salida: 3
  ```

- **clave in diccionario:** Comprueba si una clave existe en el diccionario (devuelve True o False).

  ```
  print("nombre" in persona)  # Salida: True
  print("profesion" in persona) # Salida: False
  ```

- **.keys():** Devuelve una "vista" de todas las claves.

  ```
  claves = persona.keys()
  print(claves)  # Salida: dict_keys(['nombre', 'edad', 'ciudad'])
  ```

- **.update(otro_diccionario):** Fusiona un diccionario con otro. Si hay claves repetidas, las del otro_diccionario sobrescriben a las originales.

```
info_adicional = {"profesion": "Ingeniera", "edad": 30}
persona.update(info_adicional)
print(persona)
# Salida: {'nombre': 'Ana', 'edad': 30, 'ciudad': 'Barcelona',
'profesion': 'Ingeniera'}
```

# 6. Diccionarios Anidados

Los valores de un diccionario pueden ser cualquier tipo de dato, ¡incluyendo otros diccionarios o listas! Esto permite crear estructuras de datos complejas.

```
# Lista de diccionarios
usuarios = [
    {"id": 1, "nombre": "Ana", "email": "ana@email.com"},
    {"id": 2, "nombre": "Carlos", "email": "carlos@email.com"},
    {"id": 3, "nombre": "Eva", "email": "eva@email.com"}
]

# Acceder al email del segundo usuario
print(usuarios[1]["email"])  # Salida: carlos@email.com

# Diccionario de diccionarios
empleados = {
    "emp001": {"nombre": "Luis", "departamento": "Ventas"},
    "emp002": {"nombre": "Marta", "departamento": "IT"}
}

# Acceder al departamento de Marta
print(empleados["emp002"]["departamento"]) # Salida: IT
```

# 7. Comprensión de Diccionarios (Dictionary Comprehensions)

Es una forma concisa y elegante de crear diccionarios a partir de iterables.

**Ejemplo:** Crear un diccionario con números y sus cuadrados.

```
      # Forma tradicional
cuadrados_tradicional = {}
```

```
for i in range(1, 6):
    cuadrados_tradicional[i] = i * i
print(cuadrados_tradicional)
# Salida: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Usando comprensión de diccionarios
cuadrados_comprension = {i: i * i for i in range(1, 6)}
print(cuadrados_comprension)
# Salida: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

# 8. Casos de Uso Comunes

- **Representar objetos:** Como un usuario, un producto, un coche. Es la base de formatos como JSON.
- **Contadores de frecuencia:** Contar cuántas veces aparece cada palabra en un texto.
- **Configuraciones:** Almacenar ajustes de una aplicación (ej: {'tema': 'oscuro', 'notificaciones': True}).
- **Caché:** Guardar resultados de operaciones costosas para no tener que repetirlas.

---

## Resumen Final

Los diccionarios son una herramienta fundamental en el arsenal de cualquier programador de Python. Su capacidad para asociar datos mediante claves los hace intuitivos y extremadamente eficientes para una gran variedad de problemas.

---

# 8. Funciones

## Introducción: ¿Por qué usar funciones?

Imagina que estás escribiendo un programa y te das cuenta de que necesitas realizar la misma tarea (por ejemplo, calcular el IVA de un producto) en diez lugares diferentes de tu código. Podrías copiar y pegar el mismo bloque de código diez veces, pero ¿qué pasa si necesitas cambiar el porcentaje del IVA? Tendrías que modificarlo en diez sitios distintos, lo cual es propenso a errores y muy ineficiente.

Aquí es donde entran las funciones. Una función es un **bloque de código reutilizable** que realiza una tarea específica.

**Ventajas principales:**

1. **Reutilización:** Escribes el código una vez y lo llamas cuantas veces necesites.
2. **Organización:** Dividen un programa complejo en partes más pequeñas y manejables.
3. **Abstracción:** Puedes usar una función sin necesidad de saber cómo funciona por dentro. Solo necesitas saber qué hace, qué necesita y qué devuelve.
4. **Mantenimiento:** Si necesitas hacer un cambio, solo lo haces en un lugar: dentro de la función.

---

## Capítulo 1: Lo Básico - Definiendo y Llamando una Función

La sintaxis para crear una función es muy simple. Se usa la palabra clave def, seguida del nombre de la función, paréntesis () y dos puntos :. El código que pertenece a la función debe estar indentado.

**Sintaxis:**

```
def nombre_de_la_funcion():
    # Código que ejecuta la función
    # ...
    print("¡Esta función ha sido llamada!")
```

**Llamar a una función:**
Para ejecutar el código dentro de una función, simplemente escribes su nombre seguido de paréntesis.

```
# Definimos la función
def saludar():
    print("¡Hola, mundo!")
```

```
# La llamamos para que se ejecute
saludar()
```

**Salida:**

```
¡Hola, mundo!
```

---

# Capítulo 2: Parámetros y Argumentos - Dando y Recibiendo Información

La mayoría de las veces, una función necesita recibir datos para trabajar con ellos. Estos datos se pasan a través de **parámetros**.

- **Parámetro:** Es la variable que se lista dentro de los paréntesis en la definición de la función.
- **Argumento:** Es el valor real que se envía a la función cuando se la llama.

```
# 'nombre' es un parámetro
def saludar_a(nombre):
    print(f"¡Hola, {nombre}!")

# 'Ana' y 'Carlos' son argumentos
saludar_a("Ana")
saludar_a("Carlos")
```

**Salida:**

```
¡Hola, Ana!
¡Hola, Carlos!
```

## Argumentos Posicionales vs. Argumentos de Palabra Clave (Keyword)

- **Posicionales:** Los argumentos se asignan a los parámetros según su orden.
- **De Palabra Clave (Keyword):** Especificas a qué parámetro va cada argumento, por lo que el orden no importa.

```
def describir_mascota(tipo_animal, nombre_mascota):
```

```
    print(f"Tengo un {tipo_animal} que se llama {nombre_mascota}.")

# Usando argumentos posicionales (el orden importa)
describir_mascota("perro", "Fido")

# Usando argumentos de palabra clave (el orden no importa)
describir_mascota(nombre_mascota="Misi", tipo_animal="gato")
```

**Salida:**

```
Tengo un perro que se llama Fido.
Tengo un gato que se llama Misi.
```

---

# Capítulo 3: El Valor de Retorno - return

A menudo, no solo queremos que una función haga algo, sino que también nos **devuelva un resultado**. Para esto se usa la palabra clave return.

Cuando Python encuentra un return, sale inmediatamente de la función y devuelve el valor especificado.

```
def sumar(a, b):
    resultado = a + b
    return resultado

# Guardamos el valor devuelto en una variable
total = sumar(5, 3)
print(f"El resultado de la suma es: {total}")
```

**Salida:**

```
El resultado de la suma es: 8
```

**Nota:** Si una función no tiene una declaración return, implícitamente devuelve None.

## Devolver Múltiples Valores

Una función en Python puede devolver varios valores a la vez. En realidad, los empaqueta en una **tupla**.

```
def operaciones_basicas(a, b):
```

```
    suma = a + b
    resta = a - b
    return suma, resta # Devuelve una tupla (suma, resta)

# Desempaquetamos la tupla en variables separadas
resultado_suma, resultado_resta = operaciones_basicas(10, 4)

print(f"Suma: {resultado_suma}")
print(f"Resta: {resultado_resta}")
```

**Salida:**

```
      Suma: 14
Resta: 6
```

---

# Capítulo 4: Argumentos Avanzados

## Valores por Defecto

Puedes asignar un valor por defecto a un parámetro. Si no se proporciona un argumento para ese parámetro al llamar a la función, se usará el valor por defecto.

```
def saludar_formal(nombre, saludo="Estimado/a"):
    print(f"{saludo} {nombre}, le damos la bienvenida.")

saludar_formal("Dr. Latorre")
saludar_formal("Ana", "Querida") # Sobrescribimos el valor por defecto
```

**Salida:**

```
Estimado/a Dr. Latorre, le damos la bienvenida.
Querida Ana, le damos la bienvenida.
```

## Argumentos Arbitrarios: *args y **kwargs

¿Qué pasa si no sabes cuántos argumentos va a recibir tu función?

***args (Argumentos Posicionales Arbitrarios)**

Permite pasar un número variable de argumentos posicionales. Dentro de la función, args será una **tupla** que contiene todos esos argumentos.

```
def sumar_todo(*numeros):
    total = 0
    for num in numeros:
        total += num
    return total

print(sumar_todo(1, 2, 3))
print(sumar_todo(10, 20, 30, 40, 50))
```

**Salida:**

```
6
150
```

****kwargs (Argumentos de Palabra Clave Arbitrarios)**

Permite pasar un número variable de argumentos de palabra clave. Dentro de la función, kwargs será un **diccionario**.

```
def presentar_usuario(**datos_usuario):
    print("--- Ficha de Usuario ---")
    for clave, valor in datos_usuario.items():
        print(f"{clave.capitalize()}: {valor}")
    print("------------------------")

presentar_usuario(nombre="Juan", edad=30, ciudad="Madrid")
presentar_usuario(nombre="Laura", profesion="Ingeniera")
```

**Salida:**

```
--- Ficha de Usuario ---
```

```
Nombre: Juan
Edad: 30
Ciudad: Madrid
------------------------
--- Ficha de Usuario ---
Nombre: Laura
Profesion: Ingeniera
------------------------
```

---

# Capítulo 5: Ámbito de las Variables (Scope)

El "scope" o ámbito de una variable determina desde dónde se puede acceder a ella.

- **Ámbito Local (Local Scope):** Las variables creadas dentro de una función solo existen dentro de esa función.
- **Ámbito Global (Global Scope):** Las variables creadas fuera de cualquier función son globales y se pueden leer desde cualquier parte del código.

```
variable_global = "Soy global"

def mi_funcion():
    variable_local = "Soy local"
    print(variable_local)
    print(variable_global) # Podemos leer la global desde aquí

mi_funcion()
print(variable_global)
# print(variable_local) # Esto daría un NameError, porque no existe fuera de la función
```

Para **modificar** una variable global desde una función, debes usar la palabra clave global. Sin embargo, su uso se desaconseja porque puede hacer el código difícil de seguir.

```
contador = 0

def incrementar():
    global contador
    contador += 1
    print(f"Dentro de la función: {contador}")

incrementar()
incrementar()
print(f"Fuera de la función: {contador}")
```

**Salida:**

```
Dentro de la función: 1
Dentro de la función: 2
Fuera de la función: 2
```

---

# Capítulo 6: Docstrings y Anotaciones de Tipo

## Docstrings

Es una buena práctica documentar tus funciones. Un "docstring" es una cadena de texto que se coloca como primera línea dentro de una función. Describe lo que hace la función.

```
def calcular_area_circulo(radio):
    """
    Calcula el área de un círculo dado su radio.

    Args:
        radio (float): El radio del círculo.

    Returns:
        float: El área del círculo.
    """
```

```
    pi = 3.14159
    return pi * (radio ** 2)

# Puedes acceder al docstring con la función help()
help(calcular_area_circulo)
```

### Anotaciones de Tipo (Type Hinting)

Desde Python 3.5, puedes "anotar" los tipos de los parámetros y el valor de retorno. Esto no obliga a Python a cumplir los tipos, pero ayuda a la legibilidad y a herramientas de análisis de código.

```
def saludar_con_tipos(nombre: str, edad: int) -> str:
    """Función con anotaciones de tipo."""
    return f"Hola, {nombre}. Tienes {edad} años."

saludo = saludar_con_tipos("Elena", 25)
print(saludo)
```

---

## Capítulo 7: Funciones Lambda (Funciones Anónimas)

Una función lambda es una pequeña función anónima (sin nombre) definida con la palabra clave lambda.

**Sintaxis:** lambda argumentos: expresion

- Pueden tener cualquier número de argumentos, pero solo una única expresión.
- El resultado de la expresión es lo que devuelven.

Son útiles cuando necesitas una función rápida para un uso corto, como argumento de otra función (sorted, map, filter).

```
# Función normal
def cuadrado(x):
    return x * x

# Función lambda equivalente
cuadrado_lambda = lambda x: x * x

print(cuadrado(5))
```

```
print(cuadrado_lambda(5))

# Uso práctico con sorted() para ordenar por el segundo elemento de una tupla
puntos = [(1, 5), (3, 2), (5, 9)]
puntos_ordenados = sorted(puntos, key=lambda punto: punto[1])
print(puntos_ordenados)
```

**Salida:**

```
25
25
[(3, 2), (1, 5), (5, 9)]
```

---

# Capítulo 8: Conceptos Avanzados

## Recursividad

Una función es recursiva si se llama a sí misma. Es fundamental que tenga un **caso base** para detener la recursión y evitar un bucle infinito.

Ejemplo clásico: factorial de un número.

```
def factorial(n):
    # Caso base: si n es 1, el factorial es 1
    if n == 1:
        return 1
    # Caso recursivo: n * factorial(n-1)
    else:
        return n * factorial(n - 1)

print(factorial(5)) # 5 * 4 * 3 * 2 * 1
```

## Funciones como Ciudadanos de Primera Clase

En Python, las funciones son objetos. Esto significa que puedes:

1. Asignarlas a una variable.
2. Pasarlas como argumento a otra función.
3. Devolverlas desde otra función.

```
def gritar(texto):
    return texto.upper() + "!!!"

def susurrar(texto):
    return texto.lower() + "..."

# 1. Asignar a una variable
hablar = gritar
print(hablar("Hola"))

# 2. Pasar como argumento
def saludar(funcion_formato, nombre):
    print(funcion_formato(f"Hola, {nombre}"))

saludar(gritar, "Mundo")
saludar(susurrar, "Mundo")
```

**Salida:**

```
HOLA!!!
HOLA, MUNDO!!!
hola, mundo...
```

# 9. Programación Orientada a Objetos (POO)

## Índice



---

## 1. Introducción: ¿Qué es la POO y por qué usarla?

### El paradigma de la POO

La Programación Orientada a Objetos es un **paradigma de programación**, es decir, una forma de pensar y estructurar nuestro código. En lugar de centrarnos en funciones y lógica separadas, la POO nos invita a modelar el mundo real (o un problema) a través de **objetos**.

Un **objeto** es una entidad que agrupa:

- **Datos (Atributos):** Características o propiedades del objeto (ej: el color de un coche, el nombre de una persona).
- **Comportamiento (Métodos):** Acciones que el objeto puede realizar (ej: un coche puede acelerar, una persona puede saludar).

### Ventajas de la POO

- **Reutilización de código:** A través de la herencia, podemos crear nuevas clases a partir de otras existentes.
- **Organización:** El código es más fácil de entender y mantener, ya que está agrupado en entidades lógicas (objetos).
- **Mantenibilidad:** Si necesitas cambiar cómo funciona algo, solo modificas la clase correspondiente, sin afectar al resto del programa.
- **Escalabilidad:** Es más sencillo construir sistemas grandes y complejos.

## 2. Clases y Objetos: Los Pilares Fundamentales

### ¿Qué es una clase? (El plano)

Una **clase** es como un **plano o una plantilla** para crear objetos. Define qué atributos y métodos tendrán todos los objetos de ese tipo. Por ejemplo, podríamos tener una clase Coche que define que todos los coches tendrán una marca, un modelo y podrán acelerar().

### ¿Qué es un objeto? (La construcción)

Un **objeto** (también llamado **instancia**) es la **construcción real** creada a partir de una clase. Si la clase Coche es el plano, un objeto podría ser "un Toyota Corolla rojo" y otro objeto podría ser "un Ford Mustang azul". Ambos son objetos de la clase Coche, pero tienen sus propios valores para los atributos.

### Definiendo tu primera clase

La sintaxis para crear una clase en Python es muy simple, usando la palabra clave class.

```
class Coche:
    # Atributos y métodos irán aquí
    pass # pass significa "no hacer nada", es un marcador de posición
```

### El método __init__ (El constructor)

Este es un método especial que se llama **automáticamente** cada vez que creamos un nuevo objeto de la clase. Su función principal es **inicializar los atributos** del objeto.

El primer parámetro de __init__ (y de cualquier método de instancia) es siempre self.

**¿Qué es self?**
self es una referencia **al propio objeto** que se está creando o que está llamando al método. A través de self, podemos acceder a los atributos y métodos de esa instancia específica.

```
class Coche:
    # Este es el constructor de la clase
    def __init__(self, marca, modelo, color):
        # Usamos self para asignar los valores a los atributos del objeto
        print(f"Creando un coche nuevo: {marca} {modelo}")
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.velocidad = 0 # Inicializamos la velocidad en 0
```

## Atributos: Las características del objeto

Los atributos son las variables que pertenecen a un objeto. En el ejemplo anterior, marca, modelo, color y velocidad son atributos.

## Métodos: Las acciones del objeto

Los métodos son funciones que pertenecen a una clase y definen el comportamiento de sus objetos.

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.velocidad = 0

    # Método para acelerar el coche
    def acelerar(self, cantidad):
        self.velocidad += cantidad
        print(f"El {self.marca} {self.modelo} ha acelerado. Velocidad actual:
{self.velocidad} km/h")

    # Método para frenar
    def frenar(self, cantidad):
        if self.velocidad - cantidad < 0:
```

```
            self.velocidad = 0
        else:
            self.velocidad -= cantidad
        print(f"El {self.marca} {self.modelo} ha frenado. Velocidad actual: 
{self.velocidad} km/h")

    # Método para mostrar la información del coche
    def mostrar_info(self):
        print(f"--- Ficha del Vehículo ---")
        print(f"Marca: {self.marca}")
        print(f"Modelo: {self.modelo}")
        print(f"Color: {self.color}")
        print(f"Velocidad: {self.velocidad} km/h")
```

## Creando y usando objetos

Ahora que tenemos nuestra clase Coche, podemos crear objetos (instancias) y usar sus métodos.

```
# Crear el primer objeto (instancia) de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "Rojo")

# Crear otro objeto
coche_de_ana = Coche("Ford", "Mustang", "Azul")

# Usar los métodos de los objetos
mi_coche.acelerar(50)       # Salida: El Toyota Corolla ha acelerado. Velocidad 
actual: 50 km/h
coche_de_ana.acelerar(120)  # Salida: El Ford Mustang ha acelerado. Velocidad 
actual: 120 km/h

mi_coche.frenar(20)         # Salida: El Toyota Corolla ha frenado. Velocidad 
actual: 30 km/h

# Mostrar la información de cada coche
mi_coche.mostrar_info()
# --- Ficha del Vehículo ---
# Marca: Toyota
```

```
# Modelo: Corolla
# Color: Rojo
# Velocidad: 30 km/h

coche_de_ana.mostrar_info()
# --- Ficha del Vehículo ---
# Marca: Ford
# Modelo: Mustang
# Color: Azul
# Velocidad: 120 km/h
```

---

# 3. Los 4 Pilares de la POO

## 3.1 Encapsulación: Protegiendo nuestros datos

La encapsulación consiste en **agrupar datos (atributos) y los métodos que operan sobre ellos dentro de una misma unidad (la clase)**. Además, busca controlar el acceso a esos datos para prevenir modificaciones no deseadas.

En Python, la encapsulación se logra por convención:

- **Atributos públicos:** Accesibles desde cualquier lugar (ej: self.marca).
- **Atributos "protegidos":** Se marcan con un guion bajo (_). Es una señal para otros programadores de que no deberían modificar este atributo directamente desde fuera de la clase (ej: self._voltaje).
- **Atributos "privados":** Se marcan con dos guiones bajos (__). Python cambia el nombre de estos atributos (_NombreClase__nombreAtributo), haciendo muy difícil acceder a ellos desde fuera (ej: self.__numero_serie).

```
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular  # Atributo público
        self.__saldo = saldo_inicial  # Atributo privado

    # Método público para depositar dinero (controla el acceso al saldo)
    def depositar(self, cantidad):
        if cantidad > 0:
```

```
            self.__saldo += cantidad
            print(f"Depósito exitoso. Saldo actual: ${self.__saldo}")
        else:
            print("La cantidad a depositar debe ser positiva.")

    # Método público para retirar dinero
    def retirar(self, cantidad):
        if 0 < cantidad <= self.__saldo:
            self.__saldo -= cantidad
            print(f"Retiro exitoso. Saldo actual: ${self.__saldo}")
        else:
            print("Cantidad inválida o saldo insuficiente.")

    # Método para consultar el saldo (getter)
    def consultar_saldo(self):
        return self.__saldo

# Uso
mi_cuenta = CuentaBancaria("Juan Pérez", 1000)
print(f"Titular: {mi_cuenta.titular}")

# No podemos acceder directamente a __saldo
# print(mi_cuenta.__saldo) # Esto daría un AttributeError

# Usamos los métodos públicos para interactuar con el saldo
mi_cuenta.depositar(500)
mi_cuenta.retirar(200)
print(f"El saldo final de la cuenta es: ${mi_cuenta.consultar_saldo()}")
```

## 3.2 Herencia: Reutilizando y especializando código

La herencia permite a una nueva clase (**clase hija** o **subclase**) "heredar" los atributos y métodos de una clase existente (**clase padre** o **superclase**). Esto fomenta la reutilización de código.

La clase hija puede:

- Usar los métodos y atributos de la clase padre tal como están.

- Añadir nuevos métodos y atributos.
- **Sobrescribir** métodos del padre para darles un comportamiento especializado.

```
# Clase Padre (o Superclase)
class Vehiculo:
    def __init__(self, marca):
        self.marca = marca

    def conducir(self):
        print(f"Conduciendo un vehículo de marca {self.marca}.")

# Clase Hija (hereda de Vehiculo)
# Se especifica la clase padre entre paréntesis
class CocheElectrico(Vehiculo):
    def __init__(self, marca, capacidad_bateria):
        # super() llama al constructor de la clase padre (Vehiculo)
        super().__init__(marca)
        self.capacidad_bateria = capacidad_bateria

    # Método propio de la clase hija
    def cargar_bateria(self):
        print("Cargando la batería...")

    # Sobrescribimos el método conducir de la clase padre
    def conducir(self):
        print(f"Conduciendo silenciosamente un coche eléctrico {self.marca}.")

# Uso
vehiculo_generico = Vehiculo("Genérica")
mi_tesla = CocheElectrico("Tesla", "100 kWh")

vehiculo_generico.conducir()  # Salida: Conduciendo un vehículo de marca
Genérica.
mi_tesla.conducir()          # Salida: Conduciendo silenciosamente un coche
eléctrico Tesla. (Método sobrescrito)
mi_tesla.cargar_bateria()    # Salida: Cargando la batería... (Método propio)
```

## 3.3 Polimorfismo: Objetos que toman muchas formas

El polimorfismo (del griego "muchas formas") significa que objetos de diferentes clases pueden responder al mismo mensaje (es decir, a la misma llamada de método). En Python, esto se logra a menudo a través de "Duck Typing": *Si camina como un pato y grazna como un pato, entonces es un pato.*

No importa de qué clase sea el objeto, siempre que tenga el método que estamos intentando llamar, funcionará.

```
class Perro:
    def hablar(self):
        return "¡Guau!"

class Gato:
    def hablar(self):
        return "¡Miau!"

class Pato:
    def hablar(self):
        return "¡Cuac!"

# Esta función no necesita saber de qué clase es el objeto 'animal'.
# Solo le importa que tenga un método 'hablar()'.
def hacer_hablar_a_animal(animal):
    print(animal.hablar())

# Creamos objetos de diferentes clases
perro = Perro()
gato = Gato()
pato = Pato()

# Llamamos a la misma función con diferentes objetos
hacer_hablar_a_animal(perro) # Salida: ¡Guau!
hacer_hablar_a_animal(gato)  # Salida: ¡Miau!
hacer_hablar_a_animal(pato)  # Salida: ¡Cuac!
```

## 3.4 Abstracción: Ocultando la complejidad

La abstracción consiste en **ocultar los detalles complejos de implementación** y mostrar solo la funcionalidad esencial al usuario. Una **clase abstracta** es una clase que no puede ser instanciada directamente y que define una estructura común para sus subclases.

En Python, usamos el módulo abc (Abstract Base Classes) para crear clases abstractas.

```
from abc import ABC, abstractmethod


# Clase abstracta que define una "interfaz"
class Figura(ABC):
    @abstractmethod  # Este decorador obliga a las clases hijas a implementar
este método
    def area(self):
        pass


# Si no implementas 'area', Python dará un error
class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio


    def area(self):
        return 3.14159 * (self.radio ** 2)


class Cuadrado(Figura):
    def __init__(self, lado):
        self.lado = lado


    def area(self):
        return self.lado * self.lado


# No puedes crear un objeto de una clase abstracta
# figura_generica = Figura() # Daría un TypeError


# Pero sí de sus clases hijas
```

```
mi_circulo = Circulo(10)
mi_cuadrado = Cuadrado(5)

print(f"Área del círculo: {mi_circulo.area()}")
print(f"Área del cuadrado: {mi_cuadrado.area()}")
```

---

# 4. Temas Avanzados

## Atributos de Clase vs. Atributos de Instancia

- **Atributo de Instancia:** Pertenece a un objeto específico. Se define dentro de **__init__** con self.. Cada objeto tiene su propia copia (ej: self.marca).
- **Atributo de Clase:** Pertenece a la clase en su conjunto. Es compartido por todos los objetos de esa clase. Se define directamente dentro de la clase, fuera de cualquier método.

```
class Humano:
    # Atributo de clase: es el mismo para todos los humanos
    especie = "Homo sapiens"

    def __init__(self, nombre):
        # Atributo de instancia: es único para cada humano
        self.nombre = nombre

# Uso
persona1 = Humano("Alice")
persona2 = Humano("Bob")

print(f"{persona1.nombre} es de la especie {persona1.especie}") # Alice es de la especie Homo sapiens
print(f"{persona2.nombre} es de la especie {persona2.especie}") # Bob es de la especie Homo sapiens

# Si cambias el atributo de clase, cambia para todos
Humano.especie = "Homo sapiens sapiens"
```

```
print(f"Especie actualizada para {persona1.nombre}: {persona1.especie}") # Homo
sapiens sapiens
```

## Métodos de Clase y Métodos Estáticos

- **Método de Instancia:** Es el más común. Recibe self como primer parámetro y opera sobre una instancia específica.
- **Método de Clase:** Recibe la clase (cls) como primer parámetro. Se usa para operaciones que involucran a la clase en sí, no a una instancia. Se decora con @classmethod.
- **Método Estático:** No recibe ni self ni cls. Es básicamente una función normal que vive dentro del "espacio de nombres" de la clase. Se usa para funcionalidades relacionadas con la clase pero que no dependen ni de la clase ni de una instancia. Se decora con @staticmethod.

```
class Calculadora:
    def __init__(self, propietario):
        self.propietario = propietario # Atributo de instancia

    # Método de instancia
    def sumar(self, a, b):
        print(f"{self.propietario} está sumando...")
        return a + b

    # Método de clase (un "factory method" es un uso común)
    @classmethod
    def crear_calculadora_para_empresa(cls, nombre_empresa):
        return cls(propietario=f"Empresa {nombre_empresa}")

    # Método estático (función de utilidad)
    @staticmethod
    def es_numero_par(num):
        return num % 2 == 0

# Uso
calc_personal = Calculadora("Miguel")
print(calc_personal.sumar(5, 3)) # Miguel está sumando... -> 8

# Usamos el método de clase para crear una instancia de forma especial
```

```
calc_empresa = Calculadora.crear_calculadora_para_empresa("ACME")
print(calc_empresa.propietario) # Empresa ACME

# Usamos el método estático sin crear una instancia
print(Calculadora.es_numero_par(10)) # True
```

---

# 5. Proyecto Práctico: Sistema Simple de una Biblioteca

Vamos a combinar todo lo aprendido en un pequeño proyecto.

```
class Libro:
    """Representa un libro con título, autor y estado de préstamo."""
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
        self.prestado = False # Por defecto, un libro no está prestado

    def mostrar_info(self):
        estado = "Disponible" if not self.prestado else "Prestado"
        print(f'"{self.titulo}" por {self.autor} - Estado: {estado}')

class Biblioteca:
    """Gestiona una colección de libros y las operaciones de préstamo."""
    def __init__(self, nombre):
        self.nombre = nombre
        self.__libros = [] # Encapsulación: lista de libros privada

    def agregar_libro(self, libro):
        self.__libros.append(libro)
        print(f'Libro "{libro.titulo}" agregado a la biblioteca {self.nombre}.')

    def buscar_libro(self, titulo):
        for libro in self.__libros:
            if libro.titulo.lower() == titulo.lower():
                return libro
```

```
        return None

    def prestar_libro(self, titulo):
        libro = self.buscar_libro(titulo)
        if libro:
            if not libro.prestado:
                libro.prestado = True
                print(f'El libro "{libro.titulo}" ha sido prestado.')
            else:
                print(f'El libro "{libro.titulo}" ya está prestado.')
        else:
            print(f'El libro "{titulo}" no se encuentra en la biblioteca.')

    def devolver_libro(self, titulo):
        libro = self.buscar_libro(titulo)
        if libro:
            if libro.prestado:
                libro.prestado = False
                print(f'El libro "{libro.titulo}" ha sido devuelto.')
            else:
                print(f'El libro "{libro.titulo}" no estaba prestado.')
        else:
            print(f'El libro "{titulo}" no pertenece a esta biblioteca.')

    def mostrar_catalogo(self):
        print(f"\n--- Catálogo de la Biblioteca {self.nombre} ---")
        if not self.__libros:
            print("No hay libros en el catálogo.")
        for libro in self.__libros:
            libro.mostrar_info()

# --- Simulación de uso ---

# 1. Crear la biblioteca
biblioteca_municipal = Biblioteca("Municipal")
```

```
# 2. Crear libros y agregarlos
libro1 = Libro("Cien Años de Soledad", "Gabriel García Márquez")
libro2 = Libro("Don Quijote de la Mancha", "Miguel de Cervantes")
libro3 = Libro("El Principito", "Antoine de Saint-Exupéry")

biblioteca_municipal.agregar_libro(libro1)
biblioteca_municipal.agregar_libro(libro2)
biblioteca_municipal.agregar_libro(libro3)

# 3. Mostrar el catálogo inicial
biblioteca_municipal.mostrar_catalogo()

# 4. Realizar operaciones
print("\n--- Operaciones de Préstamo ---")
biblioteca_municipal.prestar_libro("Don Quijote de la Mancha")
biblioteca_municipal.prestar_libro("Libro Inexistente")
biblioteca_municipal.prestar_libro("Don Quijote de la Mancha") # Intentar
prestar de nuevo

# 5. Mostrar el catálogo actualizado
biblioteca_municipal.mostrar_catalogo()

# 6. Devolver un libro
print("\n--- Operaciones de Devolución ---")
biblioteca_municipal.devolver_libro("Don Quijote de la Mancha")
biblioteca_municipal.mostrar_catalogo()
```

---

# 10. Tratamiento de Ficheros

## Introducción

El tratamiento de ficheros es una de las tareas más fundamentales en programación. Permite a nuestros programas leer datos de archivos, guardar resultados, configurar aplicaciones, registrar eventos (logs) y mucho más. Python ofrece una interfaz sencilla y potente para interactuar con el sistema de ficheros.

Este manual te guiará a través de todo lo que necesitas saber para dominar la manipulación de ficheros en Python.

---

## 1. Apertura y Cierre de Ficheros

La base de toda operación con ficheros es abrirlos y, muy importante, cerrarlos correctamente.

### La Función open()

Para abrir un fichero, utilizamos la función nativa open(). Sus dos parámetros más importantes son:

1. file: La ruta al fichero que queremos abrir (ej: "datos.txt" o "C:/documentos/informe.csv").
2. mode: Un string que indica el modo de apertura.

**Modos de Apertura Principales:**

| Modo | Descripción | Puntero | ¿Crea el fichero si no existe? |
|---|---|---|---|
| 'r' | **Lectura (Read)**. Es el modo por defecto. Lanza un error si el fichero no existe. | Inicio | No |
| 'w' | **Escritura (Write)**. Sobrescribe el fichero si existe. | Inicio | Sí |
| 'a' | **Añadir (Append)**. Agrega contenido al final del fichero. | Final | Sí |
| 'x' | **Creación Exclusiva (eXclusive creation)**. Crea un fichero nuevo. Lanza un error si ya existe. | Inicio | Sí |
| '+' | Se puede añadir a los modos anteriores ('r+', 'w+', 'a+') para permitir **lectura y escritura** a la vez. | | |

**Modos Binarios vs. Texto:**
Por defecto, los ficheros se abren en **modo texto**. Esto significa que Python manejará la codificación de caracteres (por ejemplo, a UTF-8). Si necesitas trabajar con ficheros que no son de texto (imágenes, ejecutables, etc.), debes añadir una 'b' al modo ('rb', 'wb').

## La Forma Correcta: El Bloque with

La mejor práctica para trabajar con ficheros en Python es usar el bloque with. Este se encarga de **cerrar el fichero automáticamente** al salir del bloque, incluso si ocurren errores.

```
# Sintaxis recomendada
with open('mi_fichero.txt', 'w') as f:
    # f es la variable que representa al fichero abierto
    # Aquí dentro realizamos las operaciones de escritura o lectura
    f.write('Hola, mundo!\n')
    f.write('Esta es la segunda línea.')

# En este punto, fuera del bloque "with", el fichero ya está cerrado.
```

**¿Por qué es mejor que f.close()?**
Si usaras el método tradicional, tendrías que recordar siempre llamar a f.close() y manejar las excepciones para asegurarte de que se cierre.

```
# Forma antigua (NO RECOMENDADADA)
f = open('mi_fichero.txt', 'w')
try:
    f.write('Hola, mundo!')
finally:
    # Esto asegura que el fichero se cierre siempre
    f.close()



El bloque with hace todo esto por ti de una manera mucho más limpia y segura.
```

---

# 2. Leer Ficheros

Una vez abierto un fichero en modo lectura ('r'), tenemos varios métodos para acceder a su contenido.

**fichero poema.txt para los ejemplos:**

```
      En el corazón tenía
la espina de una pasión;
logré arrancármela un día:
ya no siento el corazón.
```

## Leer el Fichero Completo: read()

Lee todo el contenido del fichero y lo devuelve como una única cadena de texto.

```
with open('poema.txt', 'r', encoding='utf-8') as f:
    contenido = f.read()
    print(contenido)
```

**¡Cuidado!** Usar read() en ficheros muy grandes puede consumir mucha memoria RAM, ya que carga todo el contenido de golpe.

## Leer Línea por Línea: readline()

Lee una sola línea del fichero, desde la posición actual del puntero hasta el siguiente salto de línea (\n).

```
with open('poema.txt', 'r', encoding='utf-8') as f:
    linea1 = f.readline()
    linea2 = f.readline()
    print(f"Primera línea: {linea1.strip()}") # .strip() quita espacios y saltos
de línea
    print(f"Segunda línea: {linea2.strip()}")
```

## Leer Todas las Líneas en una Lista: readlines()

Lee todas las líneas restantes del fichero y las devuelve como una lista de strings. Cada string en la lista termina con un carácter de nueva línea (\n).

```
with open('poema.txt', 'r', encoding='utf-8') as f:
    lista_de_lineas = f.readlines()
    print(lista_de_lineas)
    # Salida: ['En el corazón tenía\n', 'la espina de una pasión;\n', ...]
```

### La Forma Más Eficiente: Iterar sobre el Fichero

La forma más "pythónica" y eficiente en memoria para leer un fichero línea por línea es iterar directamente sobre el objeto del fichero.

```
with open('poema.txt', 'r', encoding='utf-8') as f:
    print("Iterando sobre el fichero:")
    for i, linea in enumerate(f):
        print(f"Línea {i+1}: {linea.strip()}")
```

```
Este método es ideal para procesar ficheros grandes porque solo carga una línea
en memoria cada vez.
```

---

## 3. Escribir en Ficheros

Para escribir, abrimos el fichero en modo 'w' (sobrescribir) o 'a' (añadir).

### Escribir una Cadena: write()

Escribe una cadena de texto en el fichero. **Importante:** write() no añade automáticamente un salto de línea. Debes añadirlo tú mismo con \n.

```
lineas_a_escribir = ["Primer renglón", "Segundo renglón", "Tercer renglón"]

with open('mi_lista.txt', 'w', encoding='utf-8') as f:
    for linea in lineas_a_escribir:
        f.write(linea + '\n') # Añadimos el salto de línea manualmente

print("Fichero 'mi_lista.txt' creado.")
```

### Escribir una Lista de Cadenas: writelines()

Escribe los elementos de una lista (u otro iterable) en el fichero. **Importante:** Al igual que write(), tampoco añade saltos de línea entre los elementos.

```
lineas_con_salto = ["Línea 1\n", "Línea 2\n", "Línea 3\n"]

with open('otro_fichero.txt', 'w', encoding='utf-8') as f:
    f.writelines(lineas_con_salto) # Las líneas ya deben tener el \n

print("Fichero 'otro_fichero.txt' creado.")
```

---

# 4. Gestión de Rutas con os.path y pathlib

Manipular rutas de ficheros como simples strings puede llevar a errores, especialmente en diferentes sistemas operativos (Windows usa \ mientras que Linux/macOS usan /).

## El Módulo os (Tradicional)

El módulo os proporciona herramientas para interactuar con el sistema operativo.

```
      import os

# Unir rutas de forma segura (la mejor manera)
ruta_base = "documentos"
nombre_fichero = "informe.txt"
ruta_completa = os.path.join(ruta_base, nombre_fichero)
print(f"Ruta segura: {ruta_completa}") # Salida: documentos/informe.txt (o
documentos\informe.txt en Windows)

# Comprobar si un fichero o directorio existe
if os.path.exists(ruta_completa):
    print(f"El fichero '{ruta_completa}' existe.")
else:
    print(f"El fichero '{ruta_completa}' no existe.")

# Otras funciones útiles
# os.remove("fichero_a_borrar.txt") # Borrar un fichero
# os.rename("antiguo.txt", "nuevo.txt") # Renombrar
# os.mkdir("nueva_carpeta") # Crear un directorio
```

## El Módulo pathlib (Moderno y Recomendado)

A partir de Python 3.4, el módulo pathlib ofrece una forma orientada a objetos para manejar rutas, que resulta más intuitiva.

```
from pathlib import Path

# Crear un objeto Path
ruta_base = Path("documentos")
nombre_fichero = "informe.txt"

# Unir rutas con el operador /
ruta_completa = ruta_base / nombre_fichero
print(f"Ruta con pathlib: {ruta_completa}")

# Comprobar si existe
if ruta_completa.exists():
    print(f"El fichero '{ruta_completa}' existe.")
```

```
# Escribir y leer texto fácilmente
ruta_completa.write_text("Este es un informe.", encoding='utf-8')
contenido = ruta_completa.read_text(encoding='utf-8')
print(f"Contenido leído con pathlib: {contenido}")

# Otras operaciones
# ruta_completa.unlink() # Borrar el fichero
# ruta_completa.rename("nuevo_informe.txt") # Renombrar
```

---

# 5. Buenas Prácticas y Consejos Adicionales

## Especificar la Codificación (encoding)

Los problemas de codificación son una fuente común de errores, especialmente al trabajar con textos que contienen acentos o caracteres especiales (como en español). Siempre es una buena práctica especificar la codificación. **utf-8** es el estándar más universal hoy en día.

```
# BUENA PRÁCTICA
with open('datos.txt', 'r', encoding='utf-8') as f:
    contenido = f.read()

# MALA PRÁCTICA (depende de la configuración por defecto del sistema)
# with open('datos.txt', 'r') as f:
#     ...
```

## Manejo de Errores

¿Qué pasa si el fichero que intentas leer no existe? El programa se detendrá con un FileNotFoundError. Puedes manejar esto con un bloque try...except.

```
from pathlib import Path

ruta_fichero = Path("fichero_que_no_existe.txt")

try:
    with open(ruta_fichero, 'r', encoding='utf-8') as f:
        print(f.read())
except FileNotFoundError:
    print(f"Error: El fichero en la ruta '{ruta_fichero}' no se ha encontrado.")
except Exception as e:
    print(f"Ha ocurrido un error inesperado: {e}")
```

## Ejemplo Práctico: Procesar un Fichero CSV

Supongamos que tenemos un fichero datos.csv con el siguiente contenido:

```
Nombre,Edad,Ciudad
Ana,28,Madrid
Luis,34,Barcelona
Carla,22,Valencia
```

Podemos leerlo y procesarlo así:

```
 import csv

# Aunque podríamos hacerlo manualmente, el módulo csv es ideal para esto.
# Aquí lo hacemos manualmente para ilustrar el manejo de ficheros.

usuarios = []
try:
    with open('datos.csv', 'r', encoding='utf-8') as f:
        next(f) # Saltamos la primera línea (cabecera)
        for linea in f:
            # Limpiamos y separamos los valores por la coma
            nombre, edad, ciudad = linea.strip().split(',')

            # Guardamos los datos en una estructura más útil
            usuarios.append({
                'nombre': nombre,
                'edad': int(edad), # Convertimos la edad a número
                'ciudad': ciudad
            })
except FileNotFoundError:
    print("El fichero datos.csv no existe.")

print(usuarios)
# Salida: [{'nombre': 'Ana', 'edad': 28, 'ciudad': 'Madrid'}, ...]
```

---

# Resumen Final

- **Usa with open(...)** para garantizar que los ficheros se cierren automáticamente.
- **Especifica siempre encoding='utf-8'** para evitar problemas con caracteres especiales.
- **Usa el módulo pathlib** para una manipulación de rutas moderna, segura y legible.
- **Itera sobre el objeto del fichero** (for line in f:) para procesar ficheros grandes de forma eficiente.
- **Anticipa errores** como FileNotFoundError usando bloques try...except.
-

# 11. Cadenas de Texto (Strings)

## Índice



---

## 1. ¿Qué es una Cadena de Texto?

Una cadena de texto (o string en inglés) es una secuencia de caracteres. En Python, cualquier cosa que pongas entre comillas se considera una cadena. Pueden ser letras, números, símbolos o espacios.

```
# Esto es una cadena
"Hola, mundo!"

# También esto
'Python 3.11'

# Incluso esto
"12345"
```

---

## 2. Creación de Cadenas

Puedes crear cadenas de tres maneras principales:

- **Comillas simples ('):** mi_cadena = 'Hola'
- **Comillas dobles ("):** mi_cadena = "Hola"
- **Comillas triples (''' o """):** Se usan para cadenas de varias líneas.

```
    cadena_simple = 'Esto funciona.'
cadena_doble = "Esto también funciona."

# Puedes usar comillas simples dentro de dobles y viceversa
frase = "Él dijo: 'Python es genial'."
frase2 = 'Ella respondió: "Totalmente de acuerdo".'

# Cadenas multilínea con comillas triples
poema = """
El sol brilla en el cielo azul,
un día perfecto para programar,
con Python, todo es más cool.
"""

print(poema)
```

---

## 3. Operaciones Básicas

### Concatenación (+)

Une dos o más cadenas para formar una nueva.

```
nombre = "Ada"
apellido = "Lovelace"

# Unimos con un espacio en medio
nombre_completo = nombre + " " + apellido
print(nombre_completo)  # Salida: Ada Lovelace
```

### Repetición (*)

Repite una cadena un número determinado de veces.

```
linea = "-" * 20
print(linea)  # Salida: --------------------

risa = "ja" * 5
print(risa)   # Salida: jajajaja
```

---

## 4. Acceso a Caracteres: Indexing y Slicing

### Indexing (Índices)

Puedes acceder a un carácter individual de una cadena usando su posición (índice), que empieza en 0.

- **Índices positivos:** 0 es el primer carácter, 1 el segundo, etc.
- **Índices negativos:** -1 es el último carácter, -2 el penúltimo, etc.

```
      palabra = "Python"
# P  y  t  h  o  n
# 0  1  2  3  4  5
#-6 -5 -4 -3 -2 -1

print(palabra[0])   # Salida: P
print(palabra[3])   # Salida: h
print(palabra[-1])  # Salida: n
print(palabra[-6])  # Salida: P
```

**Slicing (Rebanado)**

Permite extraer una sub-cadena (una porción de la cadena). La sintaxis es [inicio:fin:paso].

- inicio: Índice donde empieza el corte (incluido). Si se omite, es 0.
- fin: Índice donde termina el corte (excluido). Si se omite, es hasta el final.
- paso: El intervalo para tomar caracteres. Si se omite, es 1.

```
saludo = "Hola, Mundo"

# Desde el índice 0 hasta el 3 (el 4 no se incluye)
print(saludo[0:4])  # Salida: Hola

# Desde el índice 6 hasta el final
print(saludo[6:])   # Salida: Mundo

# Desde el principio hasta el índice 4 (el 5 no se incluye)
print(saludo[:5])   # Salida: Hola,

# Toda la cadena
print(saludo[:])    # Salida: Hola, Mundo

# Con paso de 2 (toma un carácter sí, uno no)
print(saludo[::2])  # Salida: Hla ud

# Invertir una cadena (un truco muy común)
print(saludo[::-1]) # Salida: odnuM ,aloH
```

---

## 5. La Inmutabilidad de las Cadenas

**¡Este es un concepto clave!** Las cadenas en Python son **inmutables**. Esto significa que una vez que creas una cadena, **no puedes modificarla**. Cualquier operación que parezca modificar una cadena (como replace o upper) en realidad crea una **nueva cadena** con el resultado.

```
mi_cadena = "texto original"

# Esto dará un error, no se puede cambiar un carácter
# mi_cadena[0] = "T" # TypeError: 'str' object does not support item assignment

# Para "cambiarla", creamos una nueva
nueva_cadena = "T" + mi_cadena[1:]
print(nueva_cadena) # Salida: Texto original
```

## 6. Métodos Más Comunes de las Cadenas

Los métodos son funciones que pertenecen a un objeto. Se llaman con nombre_cadena.metodo().

### Cambio de Mayúsculas/Minúsculas

- upper(): Convierte toda la cadena a mayúsculas.
- lower(): Convierte toda la cadena a minúsculas.
- capitalize(): Pone la primera letra en mayúscula y el resto en minúscula.
- title(): Pone la primera letra de cada palabra en mayúscula.
- swapcase(): Invierte mayúsculas y minúsculas.

```
      texto = "eJemPLo De TeXTo"
print(texto.upper())      # Salida: EJEMPLO DE TEXTO
print(texto.lower())      # Salida: ejemplo de texto
print(texto.capitalize()) # Salida: Ejemplo de texto
print(texto.title())      # Salida: Ejemplo De Texto
print(texto.swapcase())   # Salida: EjEMplO dE tExtO
```

### Búsqueda y Conteo

- count(sub): Cuenta cuántas veces aparece una subcadena sub.
- find(sub): Devuelve el índice de la primera aparición de sub. Si no la encuentra, devuelve -1.
- index(sub): Igual que find, pero si no encuentra sub, lanza un error (ValueError).
- startswith(prefix): Devuelve True si la cadena empieza con prefix.
- endswith(suffix): Devuelve True si la cadena termina con suffix.

```
      frase = "la casa es de color azul, la ventana también es azul"
print(frase.count("azul"))      # Salida: 2
print(frase.find("color"))      # Salida: 14
print(frase.find("rojo"))       # Salida: -1
print(frase.startswith("la"))   # Salida: True
print(frase.endswith("."))      # Salida: False
```

### Limpieza y Recorte

- strip(): Elimina espacios en blanco (o caracteres especificados) al principio y al final.
- lstrip(): Elimina espacios en blanco (o caracteres) solo al principio (izquierda).
- rstrip(): Elimina espacios en blanco (o caracteres) solo al final (derecha).

```
dato_sucio = "   usuario@email.com   "
print(f"'{dato_sucio.strip()}'")  # Salida: 'usuario@email.com'
```

```
url = "www.ejemplo.com/"
print(url.rstrip('/'))          # Salida: www.ejemplo.com
```

**Reemplazo**

- replace(old, new, count): Reemplaza todas las apariciones de old con new. Opcionalmente, se puede limitar el número de reemplazos con count.

```
      mensaje = "Hola, me llamo Juan y soy de Madrid."
nuevo_mensaje = mensaje.replace("Juan", "Ana")
print(nuevo_mensaje) # Salida: Hola, me llamo Ana y soy de Madrid.

# Reemplazar solo la primera ocurrencia
texto_repetido = "uno dos uno tres"
print(texto_repetido.replace("uno", "cien", 1)) # Salida: cien dos uno tres
```

**División y Unión**

- split(separator): Divide la cadena en una lista de subcadenas, usando separator como delimitador. Si no se especifica, divide por espacios.
- join(iterable): Une los elementos de un iterable (como una lista) en una sola cadena, usando la cadena original como separador.

```
# Split
tags = "python,java,c++,go"
lista_tags = tags.split(',')
print(lista_tags)  # Salida: ['python', 'java', 'c++', 'go']

# Join (es el inverso de split)
separador = " - "
cadena_unida = separador.join(lista_tags)
print(cadena_unida) # Salida: python - java - c++ - go
```

**Verificación de Contenido**

Estos métodos devuelven True o False. Son útiles para validaciones.

- isalnum(): True si todos los caracteres son alfanuméricos (letras o números).
- isalpha(): True si todos son letras.
- isdigit(): True si todos son dígitos.
- isspace(): True si todos son espacios en blanco.
- islower(): True si todas las letras están en minúscula.
- isupper(): True si todas las letras están en mayúscula.

```
      print("Python3".isalnum()) # True
print("Python".isalpha())  # True
print("12345".isdigit())   # True
print(" ".isspace())       # True
print("hola".islower())    # True
```

---

## 7. Formateo de Cadenas (String Formatting)

Es la forma de incrustar variables y expresiones dentro de una cadena.

### f-strings (Formato Literal de Cadenas) - MÉTODO RECOMENDADO

Introducidas en Python 3.6, son la forma más moderna, legible y eficiente. Se marcan con una f antes de las comillas.

```
nombre = "Carlos"
edad = 30
profesion = "ingeniero"

# Sintaxis simple y clara
mensaje = f"Hola, soy {nombre}, tengo {edad} años y soy {profesion}."
print(mensaje)

# También puedes poner expresiones dentro de las llaves
print(f"El año que viene tendré {edad + 1} años.")
```

### Método .format()

Una forma más antigua pero todavía muy potente y usada.

```
# Por posición
plantilla = "Hola, soy {}, tengo {} años."
print(plantilla.format("Laura", 25))

# Por nombre (keyword)
plantilla_kw = "El producto '{producto}' cuesta ${precio:.2f}."
print(plantilla_kw.format(producto="Laptop", precio=1250.99))
```

### %-formatting (Estilo antiguo)

Es el método original, similar al de C. Se ve en código antiguo, pero **no se recomienda para proyectos nuevos**.

```
nombre = "David"
print("Bienvenido, %s" % nombre)
```

---

## 8. Caracteres Especiales y Secuencias de Escape

Se usan para insertar caracteres que son difíciles de escribir directamente.

- \n: Nueva línea
- \t: Tabulación
- \\: Barra invertida
- \': Comilla simple
- \": Comilla doble

```
print("Línea 1\nLínea 2")
print("Nombre:\tJuan")
print("La ruta es C:\\Users\\Public")
```

Si quieres que Python ignore las secuencias de escape, usa una **cadena cruda (raw string)**, poniendo una r antes de las comillas.

```
# La 'r' hace que \U no se interprete como una secuencia de escape
ruta = r"C:\Users\Nuevo"
print(ruta) # Salida: C:\Users\Nuevo
```

---

## 9. Funciones Útiles que Trabajan con Cadenas

Estas no son métodos (no se llaman con . sobre la cadena), sino funciones integradas de Python.

- len(cadena): Devuelve la longitud (número de caracteres) de la cadena.
- str(objeto): Convierte un objeto (como un número) a su representación en cadena.
- ord(caracter): Devuelve el código numérico (Unicode) de un carácter.
- chr(numero): Devuelve el carácter correspondiente a un código numérico.

```
print(len("Hola"))  # Salida: 4

numero = 42
cadena_numero = str(numero)
print(type(cadena_numero)) # Salida: <class 'str'>

print(ord('A')) # Salida: 65
print(chr(65))  # Salida: A
```

---

## 10. Buenas Prácticas y Resumen

- **Prefiere f-strings:** Para formatear cadenas, usa f-strings siempre que sea posible. Son más rápidas y legibles.
- **Recuerda la inmutabilidad:** No intentes modificar cadenas. En su lugar, crea nuevas cadenas a partir de las originales.
- **Usa métodos:** Aprovecha la gran cantidad de métodos de cadena para evitar reinventar la rueda. strip(), split(), y join() son tus mejores amigos.
- **Cadenas crudas para rutas:** Al trabajar con rutas de archivos en Windows, usa r"..." para evitar problemas con las barras invertidas.
- **Consistencia en las comillas:** Elige un tipo de comilla (simple o doble) y sé consistente en tu proyecto para mejorar la legibilidad.

# 12. Todo sobre Números

## Introducción

Los números son uno de los tipos de datos más fundamentales en cualquier lenguaje de programación, y Python no es la excepción. Este manual te guiará a través de los diferentes tipos de números, las operaciones que puedes realizar con ellos y las herramientas que Python ofrece para cálculos más complejos.

---

### Índice



---

## 1. Tipos de Datos Numéricos

Python maneja principalmente tres tipos de datos numéricos.

### 1.1. Enteros (int)

Son números completos, sin parte decimal, tanto positivos como negativos. En Python 3, los enteros tienen precisión ilimitada, lo que significa que pueden ser tan grandes como la memoria de tu máquina lo permita.

```
# Ejemplos de enteros

edad = 30

temperatura = -4

poblacion_mundial = 7_900_000_000  # Los guiones bajos se pueden usar para
mejorar la legibilidad
```

```
print(type(edad))  # <class 'int'>
```

## 1.2. Números de Punto Flotante (float)

Son números que tienen una parte decimal. Se utilizan para representar números reales y se pueden escribir en notación decimal o científica.

```
# Ejemplos de floats
pi = 3.14159
precio = 19.99
distancia_estelar = 1.5e11  # Notación científica: 1.5 * 10^11

print(type(pi))  # <class 'float'>
```

## 1.3. Números Complejos (complex)

Son números con una parte real y una parte imaginaria, denotada con una j. Son menos comunes en la programación general, pero esenciales en campos científicos y de ingeniería.

```
# Ejemplo de número complejo
numero_complejo = 2 + 5j

print(type(numero_complejo))  # <class 'complex'>
print("Parte real:", numero_complejo.real)    # Salida: 2.0
print("Parte imaginaria:", numero_complejo.imag) # Salida: 5.0
```

# 2. Operadores Aritméticos Básicos

Estos son los operadores que usas para realizar cálculos matemáticos.

| Operador | Nombre | Ejemplo | Resultado | Descripción |
|---|---|---|---|---|
| + | Suma | 10 + 5 | 15 | Suma dos números. |
| - | Resta | 10 - 5 | 5 | Resta un número de otro. |
| * | Multiplicación | 10 * 5 | 50 | Multiplica dos números. |
| / | División | 10 / 3 | 3.333... | Divide y **siempre** devuelve un float. |
| // | División Entera | 10 // 3 | 3 | Divide y devuelve solo la parte entera (trunca). |
| % | Módulo (Resto) | 10 % 3 | 1 | Devuelve el resto de la división. |
| ** | Exponenciación | 2 ** 3 | 8 | Eleva un número a la potencia de otro. |

```
x = 15
y = 4

print(f"{x} + {y} = {x + y}")        # 15 + 4 = 19
print(f"{x} / {y} = {x / y}")        # 15 / 4 = 3.75
print(f"{x} // {y} = {x // y}")      # 15 // 4 = 3
print(f"{x} % {y} = {x % y}")        # 15 % 4 = 3 (porque 15 = 4*3 + 3)
```

# 3. Operadores de Comparación

Se utilizan para comparar números y siempre devuelven un valor booleano (True o False).

| Operador | Descripción | Ejemplo | Resultado |
|---|---|---|---|
| == | Igual a | 5 == 5 | True |
| != | No es igual a (distinto) | 5 != 6 | True |
| < | Menor que | 5 < 10 | True |
| > | Mayor que | 10 > 5 | True |
| <= | Menor o igual que | 5 <= 5 | True |
| >= | Mayor o igual que | 10 >= 10 | True |

---

# 4. Conversión de Tipos (Type Casting)

A menudo necesitas convertir un número de un tipo a otro.

- **int(x)**: Convierte x a un entero. Si x es un float, se trunca la parte decimal (no se redondea).
- **float(x)**: Convierte x a un número de punto flotante.
- **complex(x)**: Convierte x a un número complejo con parte imaginaria cero.
- **complex(real, imag)**: Crea un número complejo.

Esto es especialmente útil cuando se trabaja con entradas de usuario, que siempre son cadenas de texto (str).

```
# Convertir float a int (truncamiento)
print(int(9.8))  # Salida: 9

# Convertir int a float
print(float(10)) # Salida: 10.0

# Convertir una cadena (string) a número para operar
entrada_usuario = "50"
costo = int(entrada_usuario) * 2
print(f"El costo total es: {costo}") # Salida: El costo total es: 100
```

---

# 5. Funciones Numéricas Incorporadas (Built-in)

Python incluye varias funciones que son muy útiles para trabajar con números sin necesidad de importar módulos adicionales.

- **abs(x)**: Devuelve el valor absoluto de x.

  ```
  print(abs(-15)) # Salida: 15
  ```

- **round(numero, ndigitos)**: Redondea numero a ndigitos decimales. Si ndigitos se omite, redondea al entero más cercano.

- **Nota**: Python 3 usa "redondeo al par más cercano" para los casos .5. round(2.5) es 2 y round(3.5) es 4.

  ```
  print(round(3.14159, 2)) # Salida: 3.14
  print(round(3.7))        # Salida: 4
  ```

- **max(a, b, ...)**: Devuelve el mayor de los argumentos.
- **min(a, b, ...)**: Devuelve el menor de los argumentos.
- **sum(iterable)**: Devuelve la suma de los elementos de un iterable (como una lista o tupla).

```
numeros = [1, 5, -2, 10, 8]
print(max(numeros)) # Salida: 10
print(min(numeros)) # Salida: -2
print(sum(numeros)) # Salida: 22
```

---

# 6. Módulos Útiles para Operaciones Numéricas

Para matemáticas más avanzadas, Python proporciona módulos especializados.

## 6.1. El Módulo math

Contiene funciones matemáticas avanzadas y constantes. Primero debes importarlo.

```
import math

# Constantes
print(math.pi)  # 3.141592653589793
print(math.e)   # 2.718281828459045

# Funciones
print(math.sqrt(81))     # Raíz cuadrada -> 9.0
print(math.ceil(4.2))    # Redondea hacia arriba (techo) -> 5
print(math.floor(4.9))   # Redondea hacia abajo (suelo) -> 4
print(math.sin(math.pi / 2)) # Funciones trigonométricas (usan radianes) -> 1.0
print(math.log10(100))   # Logaritmo en base 10 -> 2.0
```

## 6.2. El Módulo random

Se utiliza para generar números pseudoaleatorios.

```
import random

# Entero aleatorio en un rango inclusivo
dado = random.randint(1, 6)
print(f"Lanzaste un dado y salió: {dado}")

# Flotante aleatorio entre 0.0 y 1.0
probabilidad = random.random()
print(f"Probabilidad aleatoria: {probabilidad}")

# Elegir un elemento aleatorio de una lista
opciones = ["piedra", "papel", "tijera"]
eleccion = random.choice(opciones)
print(f"El ordenador elige: {eleccion}")
```

# 7. Tópicos Avanzados y Buenas Prácticas

- **Precisión de los float**: Ten cuidado al comparar floats directamente con == debido a pequeños errores de representación binaria. Es mejor comprobar si la diferencia entre ellos es muy pequeña.

  ```
  # Mal
  # if 0.1 + 0.2 == 0.3:
  #     print("Iguales") # Esto no se ejecutará

  # Bien
  if math.isclose(0.1 + 0.2, 0.3):
      print("Cercanos") # Esto sí se ejecutará
  ```

- **Operadores a nivel de bits**: Para manipulación de bajo nivel, existen operadores como & (AND), | (OR), ^ (XOR) y <<, >> (desplazamiento de bits). Son muy específicos y se usan en áreas como criptografía o programación de sistemas.
- **NumPy**: Si vas a realizar cálculos científicos, análisis de datos o álgebra lineal intensiva, la biblioteca **NumPy** es la herramienta estándar. Proporciona un objeto array mucho más potente y eficiente que las listas de Python para operaciones numéricas masivas.

---

# 13. str.format()

## 1. Introducción: ¿Qué es el formateo de cadenas?

El formateo de cadenas (o *string formatting*) es el proceso de crear una cadena de texto insertando valores de variables o expresiones en lugares específicos de una cadena plantilla.

Python ha tenido varias formas de hacer esto a lo largo de su historia:

1. **El operador %**: El método más antiguo, heredado del lenguaje C. Es funcional, pero menos flexible y legible.
2. **El método str.format()**: Introducido en Python 2.6, ofrece una sintaxis mucho más potente, flexible y explícita.
3. **Las f-strings (cadenas literales formateadas)**: Introducidas en Python 3.6, son la forma más moderna y, a menudo, la más legible y rápida.

Aunque las f-strings son la opción preferida para código nuevo, entender str.format() sigue siendo fundamental. Es extremadamente útil en situaciones donde la cadena de formato no se conoce de antemano (por ejemplo, si se carga desde un archivo de configuración o una base de datos). Además, la "mini-sintaxis" de formato es compartida con las f-strings.

Este manual se centra exclusivamente en str.format().

---

## 2. Uso Básico

La idea principal es definir una cadena "plantilla" con marcadores de posición ({}) y luego llamar al método .format() en esa cadena, pasando los valores que se insertarán en los marcadores.

### 2.1. Marcadores de Posición Implícitos

Se rellenan en el orden en que se pasan los argumentos.

```
      # Sintaxis: "plantilla {}".format(valor)
saludo = "Hola, {}. Bienvenido a {}.".format("Ana", "Python")
print(saludo)
# Salida: Hola, Ana. Bienvenido a Python.
```

### 2.2. Marcadores de Posición Posicionales

Puedes especificar el índice del argumento que quieres usar. Esto permite reutilizar argumentos o cambiar su orden.

```
# Sintaxis: "plantilla {0} {1}".format(valor0, valor1)
mensaje = "El zorro {0} salta sobre el perro {1}.".format("marrón", "perezoso")
print(mensaje)
```

```
# Salida: El zorro marrón salta sobre el perro perezoso.


# Cambiando el orden
mensaje_invertido = "El zorro {1} salta sobre el perro {0}.".format("marrón", "perezoso")
print(mensaje_invertido)
# Salida: El zorro perezoso salta sobre el perro marrón.


# Reutilizando un argumento
repetir = "{0}, {0}, ¡qué maravilla es {1}!".format("Python", "programar")
print(repetir)
# Salida: Python, Python, ¡qué maravilla es programar!
```

## 2.3. Marcadores de Posición Nombrados (Keywords)

Puedes asignar nombres a tus marcadores de posición para mayor claridad. Esto es muy recomendable en plantillas complejas.

```
      # Sintaxis: "plantilla {nombre}".format(nombre=valor)
info_producto = "Producto: {nombre}, Precio: {precio}€, Stock: {stock} unidades".format(
    nombre="Laptop Pro",
    stock=25,
    precio=1200
)
print(info_producto)
# Salida: Producto: Laptop Pro, Precio: 1200€, Stock: 25 unidades
```

## 2.4. Combinación de Posicionales y Nombrados

Es posible combinarlos, pero todos los argumentos posicionales deben ir antes que los argumentos nombrados en la llamada a .format().

```
      mensaje = "El usuario {0} compró {item} por {precio}€.".format(
    "Alex",
    item="Teclado Mecánico",
    precio=95
)
print(mensaje)
# Salida: El usuario Alex compró Teclado Mecánico por 95€.
```

# 3. El Mini-Lenguaje de Especificación de Formato

La verdadera potencia de str.format() reside en su mini-lenguaje, que permite controlar con precisión cómo se muestra un valor. La sintaxis general dentro de las llaves es:

{<nombre_o_indice>:<especificacion>}

La especificación sigue esta estructura (los corchetes indican que es opcional):
:[[fill]align][sign][#][0][width][,][.precision][type]

Veámoslo por partes.

## 3.1. Alineación, Relleno y Ancho ([fill]align y width)

- **width**: El ancho mínimo total de la cadena resultante.
- **align**:
    - <: Alinea a la izquierda (default para la mayoría de los objetos).
    - >: Alinea a la derecha (default para números).
    - ^: Centra el contenido.
- **fill**: El carácter que se usará para rellenar el espacio sobrante (el default es un espacio).

```
    # Ancho de 15 caracteres
print("'{}'".format("Python"))          # Salida: 'Python'
print("'{:15}'".format("Python"))       # Salida: 'Python         ' (alineación izquierda por defecto)
print("'{:>15}'".format("Python"))      # Salida: '         Python' (alineación derecha)
print("'{:^15}'".format("Python"))      # Salida: '    Python     ' (centrado)

# Usando un carácter de relleno
print("'{:*^15}'".format("Python"))     # Salida: '****Python*****'
print("'{:.<15}'".format("Python"))     # Salida: 'Python.........'
```

## 3.2. Formato de Números

### Signo (sign)

- +: Muestra siempre el signo (+ para positivos, - para negativos).
- -: Muestra el signo solo para números negativos (default).
- (espacio): Usa un espacio para números positivos y un - para negativos.

```
print("'{:+}' '{:+}'".format(123, -123))   # Salida: '+123' '-123'
```

```
print("'{:-}' '{:-}'".format(123, -123))   # Salida: '123' '-123'
print("'{: }' '{: }'".format(123, -123))   # Salida: ' 123' '-123'
```

**Relleno con Ceros (0)**

Si se usa 0 justo antes del ancho, se rellenará con ceros en lugar de espacios.

```
      print("{:05}".format(42))    # Salida: '00042'
print("{:+05}".format(42))   # Salida: '+0042'
```

**Separador de Miles (,)**

Muy útil para mejorar la legibilidad de números grandes.

```
print("{:,}".format(1234567890))  # Salida: '1,234,567,890'
```

**Precisión (.precision) y Tipo (type)**

- Para **números de punto flotante (float)**:
  - f: Formato de punto fijo (decimal). .precision controla los decimales.
  - e: Notación científica.
  - %: Formato de porcentaje (multiplica por 100 y añade %).
- Para **enteros (int)**:
  - d: Decimal (base 10).
  - b: Binario (base 2).
  - o: Octal (base 8).
  - x: Hexadecimal (base 16, minúsculas).
  - X: Hexadecimal (base 16, mayúsculas).

```
# Floats
numero_pi = 3.14159265
print("Pi con 2 decimales: {:.2f}".format(numero_pi))   # Salida: Pi con 2
decimales: 3.14
print("Pi con 4 decimales: {:.4f}".format(numero_pi))   # Salida: Pi con 4
decimales: 3.1416 (redondea)
print("Notación científica: {:e}".format(12345.678)) # Salida: Notación
científica: 1.234568e+04
```

```
# Porcentaje
proporcion = 0.853
print("Completado al: {:.1%}".format(proporcion))      # Salida: Completado al: 85.3%

# Enteros en otras bases
num = 255
print("Decimal: {0:d}, Binario: {0:b}, Octal: {0:o}, Hex: {0:x}".format(num))
# Salida: Decimal: 255, Binario: 11111111, Octal: 377, Hex: ff

# El prefijo de base (#)
print("Hex con prefijo: {:#x}".format(num)) # Salida: Hex con prefijo: 0xff
```

**Truncar Cadenas**

La precisión también se puede usar en cadenas para limitar su longitud máxima.

```
frase_larga = "Esta es una frase muy larga que queremos truncar."
print("'{:.20}'".format(frase_larga))
# Salida: 'Esta es una frase mu'
```

---

# 4. Usos Avanzados

## 4.1. Acceso a Atributos de Objetos e Índices de Listas/Diccionarios

Puedes acceder directamente a los atributos de un objeto o a los elementos de una colección desde la plantilla.

```
 # Acceso a índices de una lista
datos = ["Juan", "Pérez", 35]
print("Nombre: {0[0]}, Apellido: {0[1]}, Edad: {0[2]}".format(datos))
# Salida: Nombre: Juan, Apellido: Pérez, Edad: 35
# Acceso a claves de un diccionario
persona = {"nombre": "Laura", "ciudad": "Madrid"}
print("{p[nombre]} vive en {p[ciudad]}.".format(p=persona))
# Salida: Laura vive en Madrid.

# Acceso a atributos de un objeto
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email
user = Usuario("Carlos", "carlos@email.com")
print("Usuario: {u.nombre}, Email: {u.email}".format(u=user))
# Salida: Usuario: Carlos, Email: carlos@email.com
```

## 4.2. Especificadores de Formato Anidados

Puedes usar el valor de una variable para definir una parte de la especificación de formato.

```
      ancho_dinamico = 15
precision_dinamica = 3
valor = 12.34567
# Usamos variables para definir el ancho y la precisión
plantilla = "Resultado: {val:{w}.{p}f}"
print(plantilla.format(val=valor, w=ancho_dinamico, p=precision_dinamica))
# Salida: Resultado:          12.346
```

Esto es extremadamente potente para generar tablas o reportes con alineación dinámica.

---

# 5. str.format() vs. f-strings: ¿Cuándo usar cuál?

| Característica | str.format() | f-string (f"") |
| --- | --- | --- |
| **Sintaxis** | "{}".format(valor) | f"{valor}" |
| **Legibilidad** | Menos legible, separa la plantilla de los valores. | Más legible, los valores están in-situ. |
| **Velocidad** | Ligeramente más lento. | Más rápido, evaluado en tiempo de ejecución. |
| **Caso de Uso Principal** | **Plantillas dinámicas**: cuando la cadena de formato se almacena en una variable, se lee de un archivo, etc. | La mayoría de los casos en código moderno. |

**Ejemplo de la ventaja de str.format():**

Imagina que tu aplicación soporta varios idiomas y las plantillas de mensajes están en un archivo.

```
# config.py
plantillas_es = {
    "saludo": "Hola, {nombre}. Tienes {n_mensajes} mensajes nuevos."
}
plantillas_en = {
    "saludo": "Hello, {nombre}. You have {n_mensajes} new messages."
}

# main.py
# from config import plantillas_es, plantillas_en

# Lógica para elegir el idioma
idioma_actual = plantillas_es
plantilla_saludo = idioma_actual["saludo"]

# Ahora usamos la plantilla cargada
nombre_usuario = "David"
num_mensajes = 5
print(plantilla_saludo.format(nombre=nombre_usuario, n_mensajes=num_mensajes))
# Salida: Hola, David. Tienes 5 mensajes nuevos.
```

En este escenario, una f-string no funcionaría, ya que f"" debe ser un literal en el código fuente, no puede venir de una variable.

# 6. Resumen y Buenas Prácticas

1. **Prefiere f-strings** para código nuevo por su legibilidad y rendimiento.
2. **Usa str.format()** cuando necesites trabajar con plantillas de formato que no son literales de cadena (se cargan dinámicamente).
3. **Domina el mini-lenguaje de formato**: Es la clave para un formateo avanzado y se comparte con las f-strings.
4. **Usa marcadores nombrados** ({nombre}) en str.format() siempre que sea posible para hacer tu código más explícito y menos propenso a errores.
5. **Aprovecha el acceso a atributos/índices** ({obj.attr}, {dic['key']}) para mantener las plantillas limpias y directas.

# 14. Fechas y Horas

Manejar fechas y horas es una tarea fundamental en casi cualquier aplicación, desde registrar eventos en un log hasta calcular la edad de un usuario o programar tareas. Python ofrece un ecosistema robusto para esta tarea, principalmente a través de su módulo estándar datetime.

## Índice



---

### 1. Introducción: El Módulo datetime

El módulo datetime es la herramienta principal en Python para trabajar con fechas y horas. No necesitas instalar nada, ya que viene incluido en la biblioteca estándar de Python.

Para empezar a usarlo, simplemente impórtalo:

```
import datetime
```

O, más comúnmente, importa las clases específicas que necesitas:

```
from datetime import date, time, datetime, timedelta
```

---

## 2. Los Objetos Fundamentales

### date

Representa una fecha (año, mes, día), sin información de la hora.

```
# Crear un objeto date para el 25 de diciembre de 2023
fecha_navidad = date(2023, 12, 25)

print(f"Fecha: {fecha_navidad}")
print(f"Año: {fecha_navidad.year}")
print(f"Mes: {fecha_navidad.month}")
print(f"Día: {fecha_navidad.day}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

### time

Representa una hora del día (hora, minuto, segundo, microsegundo), sin información de la fecha.

```
# Crear un objeto time para las 3:30 PM
hora_reunion = time(15, 30, 0) # 15:30:00

print(f"Hora: {hora_reunion}")
print(f"Hora: {hora_reunion.hour}")
print(f"Minuto: {hora_reunion.minute}")
print(f"Segundo: {hora_reunion.second}")
```

### datetime

Es el objeto más completo, ya que combina una fecha (date) y una hora (time).

```
# Crear un objeto datetime para el lanzamiento de un cohete
```

```
lanzamiento = datetime(2024, 10, 5, 9, 0, 0) # 5 de Octubre de 2024 a las
09:00:00

print(f"Lanzamiento: {lanzamiento}")
print(f"Fecha del lanzamiento: {lanzamiento.date()}")
print(f"Hora del lanzamiento: {lanzamiento.time()}")
print(f"Año: {lanzamiento.year}")
print(f"Minuto: {lanzamiento.minute}")
```

### timedelta

Representa una duración o la diferencia entre dos objetos date o datetime.

```
# Crear una duración de 7 días
duracion_semana = timedelta(days=7)
print(f"Una semana es: {duracion_semana}")

# Crear una duración más compleja
duracion_proyecto = timedelta(days=90, hours=8, minutes=30)
print(f"Duración del proyecto: {duracion_proyecto}")
```

---

## 3. Crear y Obtener Fechas y Horas

### Obtener la fecha y hora actual

```
# Obtener la fecha actual
hoy = date.today()
print(f"Hoy es: {hoy}")

# Obtener la fecha y hora actual (sin zona horaria)
ahora = datetime.now()
print(f"Ahora mismo: {ahora}")

# También se puede usar utcnow() para obtener la hora en UTC
ahora_utc = datetime.utcnow()
```

```
print(f"Ahora mismo en UTC: {ahora_utc}")
```

**Crear una fecha/hora específica**

Ya lo vimos antes, pero es tan simple como llamar al constructor de la clase con los valores deseados.

```
mi_cumpleaños = date(1995, 8, 15)

evento_historico = datetime(1969, 7, 20, 20, 17, 0) # Llegada a la Luna (UTC)
```

---

## 4. Formateo y Conversión (strftime & strptime)

Esta es una de las tareas más comunes: convertir fechas a texto y viceversa.

**strftime: Convertir un objeto datetime a una cadena de texto**

strftime significa "string format time". Se usa para darle un formato legible a un objeto de fecha/hora.

```
ahora = datetime.now()


# Formatos comunes

print(ahora.strftime("%Y-%m-%d"))  # Formato ISO: 2023-12-25

print(ahora.strftime("%d/%m/%Y %H:%M:%S")) # Formato común en Latam/España: 25/12/2023 15:45:30

print(ahora.strftime("Hoy es %A, %d de %B de %Y")) # Formato largo: Hoy es Lunes, 25 de Diciembre de 2023
```

**Códigos de formato más usados:**

| Código | Significado | Ejemplo |
| :--- | :--- | :--- |
| %Y | Año con 4 dígitos | 2023 |
| %y | Año con 2 dígitos | 23 |
| %m | Mes como número (01-12) | 12 |
| %B | Nombre completo del mes | Diciembre |
| %b | Nombre abreviado del mes | Dic |
| %d | Día del mes (01-31) | 25 |
| %A | Nombre completo del día | Lunes |
| %a | Nombre abreviado del día | Lun |
| %H | Hora (formato 24h, 00-23) | 15 |
| %I | Hora (formato 12h, 01-12) | 03 |
| %p | AM o PM | PM |

| %M | Minuto (00-59) | 30 |
| %S | Segundo (00-59) | 05 |

**strptime: Convertir una cadena de texto a un objeto datetime**

strptime significa "string parse time". Hace la operación inversa: lee una cadena y la convierte en un objeto datetime, siempre que el formato coincida.

```
cadena_fecha = "2024-07-26"
fecha_objeto = datetime.strptime(cadena_fecha, "%Y-%m-%d")
print(f"Objeto datetime creado: {fecha_objeto}")

cadena_completa = "30 de Enero de 2025, 10:00 AM"
# Nota: Los nombres de mes/día dependen del idioma configurado en tu sistema
# Para que funcione consistentemente, es mejor configurar el 'locale'
# import locale
# locale.setlocale(locale.LC_TIME, 'es_ES.UTF-8')
datetime_objeto_completo = datetime.strptime(cadena_completa, "%d de %B de %Y, %I:%M %p")
print(f"Objeto completo creado: {datetime_objeto_completo}")
```

**¡Importante!** El formato que pasas a strptime debe coincidir **exactamente** con la cadena de texto, incluyendo espacios, comas y guiones.

---

## 5. Aritmética de Fechas con timedelta

### Sumar y restar tiempo

Usa timedelta para moverte hacia adelante o atrás en el tiempo.

```
ahora = datetime.now()
print(f"Ahora: {ahora}")

# ¿Qué fecha y hora será en 10 días?
futuro = ahora + timedelta(days=10)
print(f"En 10 días: {futuro}")

# ¿Qué fecha y hora era hace 2 semanas y 3 horas?
pasado = ahora - timedelta(weeks=2, hours=3)
print(f"Hace 2 semanas y 3h: {pasado}")
```

### Calcular la diferencia entre dos fechas

Restar dos objetos datetime o date te da un objeto timedelta.

```
      fecha_navidad = datetime(2024, 12, 25)
hoy = datetime.now()

diferencia = fecha_navidad - hoy

print(f"Faltan para Navidad: {diferencia}")
print(f"Días que faltan: {diferencia.days}")
print(f"Segundos totales que faltan: {diferencia.total_seconds()}")
```

---

## 6. Zonas Horarias (Timezones)

Este es un concepto avanzado pero crucial.

- **Fecha/Hora "Naive" (Ingenua):** No tiene información de zona horaria. datetime.now() crea una fecha "naive".
- **Fecha/Hora "Aware" (Consciente):** Tiene información de zona horaria y sabe en qué parte del mundo se encuentra.

Desde Python 3.9, la forma recomendada de trabajar con zonas horarias es el módulo zoneinfo.

```
from zoneinfo import ZoneInfo

# 1. Crear una fecha "naive"
dt_naive = datetime(2024, 8, 1, 12, 0, 0)

# 2. Definir zonas horarias
tz_madrid = ZoneInfo("Europe/Madrid")
tz_tokio = ZoneInfo("Asia/Tokyo")
```

```
# 3. Convertir la fecha "naive" a una "aware" (asumiendo que era de Madrid)
dt_madrid = dt_naive.replace(tzinfo=tz_madrid)
print(f"Hora en Madrid: {dt_madrid}")

# 4. Convertir a otra zona horaria
dt_tokio = dt_madrid.astimezone(tz_tokio)
print(f"Hora en Tokio:  {dt_tokio}")

# Para obtener la hora actual con zona horaria:
ahora_madrid = datetime.now(tz_madrid)
print(f"Hora actual en Madrid: {ahora_madrid}")
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. Python
IGNORE_WHEN_COPYING_END

---

## 7. Ejemplo Práctico: Calculadora de Edad

Este script pide al usuario su fecha de nacimiento y calcula su edad.

```
from datetime import datetime

# Pedir la fecha de nacimiento al usuario
str_fecha_nacimiento = input("Introduce tu fecha de nacimiento (formato DD/MM/AAAA): ")

try:
    # Convertir la cadena a un objeto date
    fecha_nacimiento = datetime.strptime(str_fecha_nacimiento, "%d/%m/%Y").date()

# Obtener la fecha de hoy
hoy = date.today()

    # Calcular la diferencia
    diferencia = hoy - fecha_nacimiento
```

```
    # Calcular los años de forma aproximada
    años = diferencia.days // 365

    # Una forma más precisa de calcular la edad
    edad = hoy.year - fecha_nacimiento.year - ((hoy.month, hoy.day) <
(fecha_nacimiento.month, fecha_nacimiento.day))

print(f"Tienes {edad} años.")

except ValueError:
    print("El formato de la fecha no es válido. Por favor, usa DD/MM/AAAA.")
```

---

## 8. Otros Módulos Útiles

**time**

Es un módulo de más bajo nivel. Es útil para:

- time.time(): Obtener el "timestamp" de Unix (segundos desde el 1 de enero de 1970). Muy usado para guardar fechas en bases de datos de forma eficiente.
- time.sleep(segundos): Pausar la ejecución de un programa durante un número de segundos.

```
import time

print("Inicio")
time.sleep(2) # Espera 2 segundos
print("Fin")

timestamp = time.time()
print(f"Timestamp actual: {timestamp}")
# Puedes convertir un timestamp a un datetime
print(f"Fecha desde timestamp: {datetime.fromtimestamp(timestamp)}")
```

**calendar**

Como su nombre indica, es útil para generar calendarios en formato de texto.

```
import calendar

# Imprimir el calendario de diciembre de 2023
cal = calendar.month(2023, 12)
print(cal)

# Comprobar si un año es bisiesto
print(f"¿2024 es bisiesto? {calendar.isleap(2024)}")
```

---

# 15. Color

## Códigos ANSI: Sintaxis general

```
\033[<estilo>;<color_texto>;<color_fondo>m
```

- `\033[` es la secuencia de escape.
- Los códigos terminan con `m`.
- Puedes combinar estilos, colores de texto y fondo.

---

## Ejemplo básico de colores

```
print("\033[31mTexto en rojo\033[0m")
print("\033[32mTexto en verde\033[0m")
print("\033[33mTexto en amarillo\033[0m")
print("\033[34mTexto en azul\033[0m")
print("\033[35mTexto en magenta\033[0m")
print("\033[36mTexto en cian\033[0m")
```

---

## Colores (texto o fondo)

| Color | Código texto | Código fondo |
|---|---|---|
| Negro | 30 | 40 |
| Rojo | 31 | 41 |
| Verde | 32 | 42 |
| Amarillo | 33 | 43 |
| Azul | 34 | 44 |
| Magenta | 35 | 45 |
| Cian | 36 | 46 |
| Blanco | 37 | 47 |

---

## Estilos

| Estilo | Código |
| --- | --- |
| Normal | 0 |
| Negrita | 1 |
| Subrayado | 4 |
| Inverso | 7 |

---

## Ejemplo combinado (negrita + rojo + fondo blanco)

```
print("\033[1;31;47m¡Hola Mundo con estilo!\033[0m")
```

---

## Importante

El `\033[0m` al final **restaura** el color por defecto. Siempre úsalo para evitar que el resto del texto quede coloreado accidentalmente.

---

# 16. Graficos

## 1. Turtle

`Turtle` es una librería muy visual y educativa de Python que permite crear gráficos y dibujos mediante instrucciones simples. Es ideal para aprender lógica de programación y conceptos de geometría o arte computacional.

---

## Configuración básica

```
import turtle

pantalla = turtle.Screen()         # Crea la ventana
pantalla.bgcolor("white")          # Fondo blanco

t = turtle.Turtle()                # Crea la "tortuga"
t.speed(3)                         # Velocidad de dibujo (1 lento, 10 rápido)
```

---

## Ejemplo 1: Dibujar un cuadrado

```
for _ in range(4):
    t.forward(100)   # Avanza 100 unidades
    t.right(90)      # Gira 90 grados a la derecha
```

---

## Ejemplo 2: Estrella de 5 puntas

```
for _ in range(5):
    t.forward(150)
    t.right(144)
```

---

# Ejemplo 3: Espiral de colores

```
import random

colores = ["red", "blue", "green", "purple", "orange", "black"]

for i in range(100):
    t.color(random.choice(colores))
    t.forward(i)
    t.right(59)
```

---

# Ejemplo 4: Roseta con bucles

```
t.pensize(2)
t.color("blue")

for i in range(36):
    for j in range(4):
        t.forward(100)
        t.right(90)
    t.right(10)
```

---

# Finalizar

Cuando termines el dibujo, puedes mantener la ventana abierta con:

```
turtle.done()
```

---

# 1. Matplotlib

---

## Paso 1: Instalar matplotlib (si no lo tienes)

```
pip install matplotlib
```

---

## Ejemplo 1: Gráfico de líneas

```
import matplotlib.pyplot as plt


x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]


plt.plot(x, y, marker='o', color='blue', linestyle='-')
plt.title("Gráfico de líneas")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.grid(True)
plt.show()
```

---

## Ejemplo 2: Gráfico de barras

```
import matplotlib.pyplot as plt


categorias = ['A', 'B', 'C', 'D']
valores = [10, 15, 7, 12]


plt.bar(categorias, valores, color='green')
plt.title("Gráfico de barras")
plt.xlabel("Categorías")
plt.ylabel("Valores")
plt.show()
```

---

## Ejemplo 3: Gráfico de barras horizontales

```
import matplotlib.pyplot as plt
```

```
categorias = ['A', 'B', 'C', 'D']
valores = [10, 15, 7, 12]

plt.barh(categorias, valores, color='purple')
plt.title("Gráfico de barras horizontales")
plt.xlabel("Valores")
plt.ylabel("Categorías")
plt.show()
```

---

# Ejemplo 4: Gráfico de pastel (pie chart)

```
import matplotlib.pyplot as plt

etiquetas = ['Manzanas', 'Peras', 'Plátanos', 'Uvas']
porcentajes = [30, 25, 25, 20]

plt.pie(porcentajes, labels=etiquetas, autopct='%1.1f%%', startangle=90)
plt.title("Distribución de frutas")
plt.axis('equal')  # Para que sea un círculo
plt.show()
```

---

# Ejemplo 5: Gráfico de dispersión (scatter plot)

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [5, 7, 4, 6, 8]

plt.scatter(x, y, color='red')
plt.title("Gráfico de dispersión")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.show()
```

---

# Extras

## Guardar un gráfico como imagen

```
plt.savefig("grafico.png")  # Guardar como PNG
```

## Múltiples gráficos (subplots)

```
fig, axs = plt.subplots(1, 2)

axs[0].plot([1, 2, 3], [1, 4, 9])
axs[0].set_title("Gráfico 1")

axs[1].bar(['A', 'B', 'C'], [5, 7, 3])
axs[1].set_title("Gráfico 2")

plt.tight_layout()
plt.show()
```

---

# 17. Caracteres ascii

En Python, puedes convertir caracteres a su valor ASCII y viceversa usando las funciones incorporadas `ord()` y `chr()`:

## 1. Convertir un carácter a su valor ASCII

```
valor_ascii = ord('A')
print(valor_ascii)  # Salida: 65
```

## 2. Convertir un valor ASCII a carácter

```
caracter = chr(65)
print(caracter)  # Salida: 'A'
```

---

## Ejemplo completo:

```
texto = "Hola"
ascii_valores = [ord(c) for c in texto]
print("Texto a ASCII:", ascii_valores)

# Volver a texto
texto_original = ''.join(chr(c) for c in ascii_valores)
print("ASCII a texto:", texto_original)
```

### Salida:

```
Texto a ASCII: [72, 111, 108, 97]
ASCII a texto: Hola
```

# Anexo

## Operaciones Básicas

```
# Aritmética
a + b       # Suma
a - b       # Resta
a * b       # Multiplicación
a / b       # División flotante
a // b      # División entera
a % b       # Módulo (resto)
a ** b      # Potencia

# Comparaciones
a == b      # Igualdad
a != b      # Desigualdad
a > b       # Mayor que
a < b       # Menor que
a >= b      # Mayor o igual que
a <= b      # Menor o igual que

# Lógicos
a and b     # Y lógico
a or b      # O lógico
not a       # Negación lógica

# Asignación
x = 5
x += 1      # Incrementa
x -= 1      # Decrementa
x *= 2      # Multiplica
x /= 2      # Divide
```

---

## Funciones de Texto

```
texto.lower()       # Minúsculas
texto.upper()       # Mayúsculas
texto.title()       # Título
texto.strip()       # Elimina espacios
texto.replace("a", "b")   # Reemplaza
texto.split(" ")     # Divide en lista
" ".join(lista)      # Une lista a string
len(texto)           # Longitud
texto.startswith("Hola")
texto.endswith("fin")
```

---

# Funciones de Listas

```
lista = [1, 2, 3]
lista.append(4)         # Añade al final
lista.insert(1, 99)     # Inserta en índice
lista.remove(2)         # Elimina el valor
valor = lista.pop()     # Quita el último
lista.sort()            # Ordena
lista.reverse()         # Invierte
len(lista)              # Longitud
min(lista), max(lista), sum(lista)
```

---

# Funciones Matemáticas

```
import math

math.sqrt(x)          # Raíz cuadrada
math.pow(x, y)        # Potencia
math.factorial(n)     # Factorial
math.sin(x), math.cos(x), math.tan(x)
math.pi               # Pi
math.e                # Número e
round(x), abs(x)
```

---

# Funciones Aleatorias

```
import random

random.random()           # [0,1)
random.randint(a, b)      # Entero entre a y b
random.choice(lista)      # Elemento aleatorio
random.shuffle(lista)     # Baraja la lista
```

---

# Funciones Generales de Python

```
print("Hola")          # Imprimir
input("Nombre: ")      # Entrada
type(x)                # Tipo de dato
isinstance(x, int)     # Verifica tipo
range(n)               # Generador de rango
enumerate(lista)       # Enumerar elementos
zip(lista1, lista2)    # Agrupar en pares
map(func, lista)       # Aplicar función
filter(cond, lista)    # Filtrar elementos
```

---

# Ficheros

```
# Leer archivo
with open("archivo.txt", "r") as f:
    contenido = f.read()

# Escribir archivo
with open("archivo.txt", "w") as f:
    f.write("Hola mundo")

# Añadir contenido
with open("archivo.txt", "a") as f:
    f.write("Nueva línea\n")
```

---

# Bucles y Condiciones

```
# Condicional
if x > 0:
    print("Positivo")
elif x == 0:
    print("Cero")
else:
    print("Negativo")

# Bucle for
for i in range(5):
    print(i)

# Bucle while
while x > 0:
    print(x)
    x -= 1
```

---

# Bucles vacios

```
for i in range(5):
      pass        #bucle vacio
```

---

# Funciones Propias

```
def saludar(nombre):
    return f"Hola, {nombre}"

resultado = saludar("Ana")
```

---

# Manejo de Errores

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("No se puede dividir entre cero")
finally:
    print("Fin del bloque")
```

---

# Borrar pantalla

```
from os import system
system(“cls”)
```

---

# Ejecutar un programa externo .exe

```
import subprocess
subprocess.run([“c:\directorio\archivo.exe”])
```

---

# Ejecutar un programa externo .py

```
With open(“archivo.py”) as f:
      exec(f.read())
```

---

# Crear un archivo .exe

```
Hay que instalar en el terminal “nueva terminal de python”:
pip install pyinstaller

Para crear el archivo ejecutable hay que poner en la terminal “nueva terminal de
python”:
pyinstaller -onefile archivo.py
```

---

# Teclas especiales

| Código | Significado |
|---|---|
| `\n` | Nueva línea |
| `\t` | Tabulación |
| `\\` | Barra invertida |
| `\'` | Comilla simple |
| `\"` | Comilla doble |
| `\r` | Retorno de carro |
| `\b` | Retroceso |

# Código ASCII

Fue definido en 1963 y ha sido adoptado como el estándar para la transmisión de datos. Este código denominado ASCII (Se pronuncia “ASKI” y viene de las siglas "American Standard Code for Information Interchange "), permite representar casi todos los caracteres diferentes.  El procedimiento es activar la tecla “ALT” + el número(s) indicado, por el teclado numérico.

## Caracteres ASCII de control

| | | |
|---|---|---|
| 00 | NULL | (carácter nulo) |
| 01 | SOH | (inicio encabezado) |
| 02 | STX | (inicio texto) |
| 03 | ETX | (fin de texto) |
| 04 | EOT | (fin transmisión) |
| 05 | ENQ | (consulta) |
| 06 | ACK | (reconocimiento) |
| 07 | BEL | (timbre) |
| 08 | BS | (retroceso) |
| 09 | HT | (tab horizontal) |
| 10 | LF | (nueva línea) |
| 11 | VT | (tab vertical) |
| 12 | FF | (nueva página) |
| 13 | CR | (retorno de carro) |
| 14 | SO | (desplaza afuera) |
| 15 | SI | (desplaza adentro) |
| 16 | DLE | (esc.vínculo datos) |
| 17 | DC1 | (control disp. 1) |
| 18 | DC2 | (control disp. 2) |
| 19 | DC3 | (control disp. 3) |
| 20 | DC4 | (control disp. 4) |
| 21 | NAK | (conf. negativa) |
| 22 | SYN | (inactividad sínc) |
| 23 | ETB | (fin bloque trans) |
| 24 | CAN | (cancelar) |
| 25 | EM | (fin del medio) |
| 26 | SUB | (sustitución) |
| 27 | ESC | (escape) |
| 28 | FS | (sep. archivos) |
| 29 | GS | (sep. grupos) |
| 30 | RS | (sep. registros) |
| 31 | US | (sep. unidades) |
| 127 | DEL | (suprimir) |

## Caracteres ASCII imprimibles

| | | | | | |
|---|---|---|---|---|---|
| 32 | espacio | 64 | @ | 96 | ` |
| 33 | ! | 65 | A | 97 | a |
| 34 | " | 66 | B | 98 | b |
| 35 | # | 67 | C | 99 | c |
| 36 | $ | 68 | D | 100 | d |
| 37 | % | 69 | E | 101 | e |
| 38 | & | 70 | F | 102 | f |
| 39 | ' | 71 | G | 103 | g |
| 40 | ( | 72 | H | 104 | h |
| 41 | ) | 73 | I | 105 | i |
| 42 | * | 74 | J | 106 | j |
| 43 | + | 75 | K | 107 | k |
| 44 | , | 76 | L | 108 | l |
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |
| 53 | 5 | 85 | U | 117 | u |
| 54 | 6 | 86 | V | 118 | v |
| 55 | 7 | 87 | W | 119 | w |
| 56 | 8 | 88 | X | 120 | x |
| 57 | 9 | 89 | Y | 121 | y |
| 58 | : | 90 | Z | 122 | z |
| 59 | ; | 91 | [ | 123 | { |
| 60 | < | 92 | \ | 124 | \| |
| 61 | = | 93 | ] | 125 | } |
| 62 | > | 94 | ^ | 126 | ~ |
| 63 | ? | 95 | _ | | |

## ASCII extendido (Página de código 437)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 135 | ç | 167 | º | 199 | Ã | 231 | þ |
| 136 | ê | 168 | ¿ | 200 | ╚ | 232 | Þ |
| 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 144 | É | 176 | ░ | 208 | ð | 240 | ≡ |
| 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | nbsp |

| | | | | | |
|---|---|---|---|---|---|
| 1 | ☺ | 12 | ♀ | 23 | ↨ |
| 2 | ☻ | 13 | ♪ | 24 | ↑ |
| 3 | ♥ | 14 | ♫ | 25 | ↓ |
| 4 | ♦ | 15 | ☼ | 26 | → |
| 5 | ♣ | 16 | ► | 27 | ← |
| 6 | ♠ | 17 | ◄ | 28 | ∟ |
| 7 | • | 18 | ↕ | 29 | ↔ |
| 8 | ◘ | 19 | ‼ | 30 | ▲ |
| 9 | ○ | 20 | ¶ | 31 | ▼ |
| 10 | ◙ | 21 | § | | |
| 11 | ♂ | 22 | ▬ | | |

## los más consultados

| | |
|---|---|
| \ | **barra invertida** (alt + 92) |
| @ | **arroba** (alt + 64) |
| ñ | **eñe minuscula** (alt + 164) |
| ' | **comilla simple, apóstrofe** (alt + 39) |
| # | **signo numeral** (alt + 35) |
| ! | **signo de admiración** (alt + 33) |
| _ | **guión bajo, subrayado** (alt + 95) |
| * | **asterisco** (alt + 42) |
| ~ | **equivalencia, tilde** (alt + 126) |
| - | **guión medio** (alt + 45) |

## de uso frecuente ( idioma español )

| | |
|---|---|
| ñ | alt + 164 |
| Ñ | alt + 165 |
| @ | alt + 64 |
| ¿ | alt + 168 |
| ? | alt + 63 |
| ¡ | alt + 173 |
| ! | alt + 33 |
| : | alt + 58 |
| / | alt + 47 |
| \ | alt + 92 |

## vocales con acento ( acento agudo español )

| | |
|---|---|
| á | alt + 160 |
| é | alt + 130 |
| í | alt + 161 |
| ó | alt + 162 |
| ú | alt + 163 |
| Á | alt + 181 |
| É | alt + 144 |
| Í | alt + 214 |
| Ó | alt + 224 |
| Ú | alt + 233 |

## vocales con diéresis

| | |
|---|---|
| ä | alt + 132 |
| ë | alt + 137 |
| ï | alt + 139 |
| ö | alt + 148 |
| ü | alt + 129 |
| Ä | alt + 142 |
| Ë | alt + 211 |
| Ï | alt + 216 |
| Ö | alt + 153 |
| Ü | alt + 154 |

## símbolos matemáticos

| | |
|---|---|
| ½ | alt + 171 |
| ¼ | alt + 172 |
| ¾ | alt + 243 |
| ¹ | alt + 251 |
| ³ | alt + 252 |
| ² | alt + 253 |
| ƒ | alt + 159 |
| ± | alt + 241 |
| × | alt + 158 |
| ÷ | alt + 246 |

## símbolos comerciales

| | |
|---|---|
| $ | alt + 36 |
| £ | alt + 156 |
| ¥ | alt + 190 |
| ¢ | alt + 189 |
| ¤ | alt + 207 |
| ® | alt + 169 |
| © | alt + 184 |
| ª | alt + 166 |
| º | alt + 167 |
| ° | alt + 248 |

## comillas, llaves paréntesis

| | |
|---|---|
| " | alt + 34 |
| ' | alt + 39 |
| ( | alt + 40 |
| ) | alt + 41 |
| [ | alt + 91 |
| ] | alt + 93 |
| { | alt + 123 |
| } | alt + 125 |
| « | alt + 174 |
| » | alt + 175 |

# Epílogo

Has llegado al final de este libro, y con ello, al comienzo de algo mucho más grande: tu camino como programador. Aprender Python te ha abierto una puerta a un universo de posibilidades. Has aprendido a pensar como un programador, a resolver problemas con lógica y creatividad, y a convertir tus ideas en código.

Pero esto es solo el principio.

La programación no es una meta, es una herramienta para explorar, crear y transformar. Con cada línea de código que escribes, estás entrenando tu mente para pensar de forma estructurada y clara. Estás construyendo puentes entre la imaginación y la realidad.

Sigue practicando. Prueba proyectos pequeños y luego sueña en grande. Participa en comunidades, explora nuevas bibliotecas, colabora con otros. Equivócate, aprende, vuelve a intentarlo. El mejor aprendizaje viene del hacer.

Recuerda: cada gran programador fue, alguna vez, un principiante que no se rindió.

Sigue programando. El mundo necesita tus ideas.

¡Nos vemos en el próximo script!